Detection and Analysis of Connection Chains in Network Forensics

by

Ahmad Almulhem

M.S., King Fahd University of Petroleum and Minerals, Saudi Arabia, 1996
B.S., King Fahd University of Petroleum and Minerals, Saudi Arabia, 1991

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

Detection and Analysis of Connection Chains in Network Forensics

by

Ahmad Almulhem

M.S., King Fahd University of Petroleum and Minerals, Saudi Arabia, 1996
B.S., King Fahd University of Petroleum and Minerals, Saudi Arabia, 1991

Supervisory Committee

Dr. I. Traore, Supervisor
(Department of Electrical and Computer Engineering)

Dr. F. Gebali, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. M. Sima, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. K. Wu, Outside Member
(Department of Computer Science)

Dr. I. Woungang, External Examiner
(Department of Computer Science, Ryerson University, Toronto, Canada)

Supervisory Committee
Dr. I. Traore, Supervisor
(Department of Electrical and Computer Engineering)

Dr. F. Gebali, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. M. Sima, Departmental Member
(Department of Electrical and Computer Engineering)

Dr. K. Wu, Outside Member
(Department of Computer Science)

Dr. I. Woungang, External Examiner
(Department of Computer Science, Ryerson University, Toronto, Canada)

ABSTRACT

Network forensics is a young member of the bigger family of digital forensics discipline. In particular, it refers to digital forensics in networked environments. It represents an important extension to the model of network security where emphasis is traditionally put on prevention and to a lesser extent on detection. It focuses on the *collection*, and *analysis* of network packets and events caused by an intruder for investigative purposes.

A key challenge in network forensics is to ensure that the network itself is forensically-ready, by providing an infrastructure to collect and analyze data in real-time. In this thesis, we propose an agent-based network forensics system, which is intended to add real-time network forensics capabilities into

a controlled network. We also evaluate the proposed system by deploying and studying it in a real-life environment.

Another challenge in network forensics arises because of attackers ability to move around in the network, which results in creating a chain of connections; commonly known as *connection chains*. In this thesis, we provide an extensive review and taxonomy of connection chains. Then, we propose a novel framework to detect them. The framework adopts a black-box approach by passively monitoring inbound and outbound packets at a host, and analyzing the observed packets using association rule mining. We assess the proposed framework using public network traces, and demonstrate both its efficiency and detection capabilities.

We, finally, propose a profiling-based framework to investigate connection chains that are distributed over several ip addresses. The framework utilizes a simple yet extensible hacker model that integrates information about a hacker's linguistic, operating system and time of activity. We establish the effectiveness of the proposed approach through several simulations and an evaluation with real attack data.

Table of Contents

List of Tables

List of Figures

Acronyms

CSI	Computer Security Institute
DDOS	Distributed Denial of Service Attack
DOS	Denial of Service Attack
FDR	False Detection Rate
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
MYSQL	Free Database Sever
NIDE	Next-Generation Intrusion-Detection Expert System
RLOGIN	Remote Login
RPC	Remote Procedure Call
SNORT	Free Intrusion Detection System
SSH	Secure Shell
TCP	Transmission Control Protocol
TDR	True Detection Rate
TELNET	Teletype Network Protocol
UDP	User Datagram Protocol

Acknowledgments

First of all, I would like to express my deepest gratitude to my thesis adviser, Dr. Issa Traore, for his help, encouragement, and financial support. Dr. Issa's dedication, knowledge, and kindness have profoundly influenced me and guided me through my Ph.D. studies.

Also, I would like to thank Dr. Fayez Gebali, Dr. Kui Wu and Dr. Mihai Sima for being on my thesis committee. Their suggestions and comments were very valuable. Special thanks go to Dr. Fayez Gebali for his kind help and support when I arrived Victoria. I also wish to thank Dr Isaac Woungang, from Ryerson University in Toronto, for kindly being the external examiner.

I also wish to thank our staff Ms. Vicky Smith, Ms. Lynne Barrett, Ms. Moneca Bracken, Ms. Mary-Anne Teo, Steve Campbell and Erik Laxdal for their great support and continuous cooperation.

I also have been lucky to meet good friends and colleges in my research group in the ISOT lab. My thanks go to Wei Lu, Akif Nazar, Ahmed Awad, Bassam Sayed and Alex Hoole, for your friendship and helpful discussions. Outside the ISOT lab, I wish to thank Khalid Khayyat, Khalid Almuzaini

and Mohamad Fayed for kindly helping me during my stay in Victoria.

Many thanks also go to my sponsor in Saudi Arabia; King Fahd University of Petroleum and Minerals (KFUPM). Thank you for sponsoring my Ph.D. studies. Also, my thanks go to the Saudi Arabian Cultural Bureau in Canada for helping me during my stay in Canada.

Finally, my special thanks go to my family in Saudi Arabia for their love, support and prayers. I like to thank my mother, sisters, brothers, nieces and nephews. Particularly, I like to thank my brother Mohammad for his kind support and follow-ups. Also, I like to thank my mother for her unconditional love and continuing prayers.

Dedication

To my mother

and

in memory of my father

Chapter 1

Introduction

1.1 Context

Undoubtedly, the Internet has revolutionized the computer and communications world like nothing before. It has opened the doors for enormous advancement and applications. Unfortunately, however, it has also allowed for great possibilities of abuse and damage.

Nowadays, it is more common than ever for computer attacks and crimes to make headlines in media. The CSI/FBI annual survey is an insightful reference of how often those crimes occur and how expensive they can be. In their survey for the year 2003 [1], it was reported that the total annual financial losses were more than $200 millions USD. This figure could actually be worse, since only 251 out of the 530 participants (47%) reported their financial losses.

The survey also shows other compelling statistics.

- 92% of the respondents detected attacks during the last 12 months.
- 75% of the respondents acknowledged financial losses due to security breaches.
- Theft of propriety information is reported to be the costliest form of computer crime with more than $70 millions USD. Denial of service attacks came next with more than $65 millions USD.

In many cases, the costly theft of propriety information is due to the classic theft of business information by insiders. The website of U.S. department of justice lists many of such cases [2]. The following case is quoted from there.

> *"Richard Glenn Dopps, 35, pleaded guilty to one felony count of obtaining information from a protected computer.*
>
> *Until February 2001, Dopps was employed by The Bergman Companies (TBC), a contracting firm based in Chino. After leaving TBC to go work for a competitor, Dopps used his Internet connection to gain access to TBCs computer systems on more than 20 occasions.*
>
> *Once Dopps was inside the TBC systems, he read e-mail messages of TBC executives to stay informed of TBCs ongoing business and to obtain a commercial advantage for his new employer.*

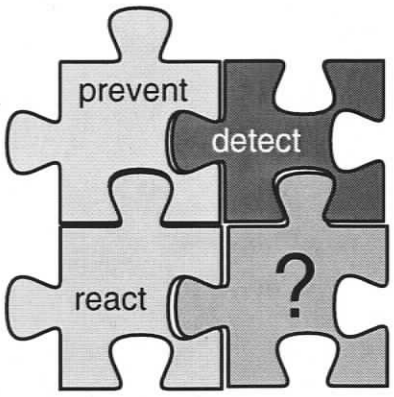

Figure 1.1: Security Model.

Dopps unauthorized access into TBCs computer system caused approximately $21,636 in damages and costs to TBC."

Unfortunately, those attacks will only get worse. Our networks are only getting larger and more complex. They are unlikely to shrink or become simpler.

Conceptually, many organizations address security from three different perspectives; namely *prevention*, *detection*, and *reaction* (Figure 1.1). According to the 2003 CSI/FBI survey, 99% of the respondents use a mixture of various technologies pertaining to those perspectives. For example, more than 90% use prevention technologies, like firewall, access control, and physical security. Also, 73% use intrusion detection systems.

1.2 Problem Statement

Although the above model is quite effective, a very important piece is missing. Specifically, current technologies lack any *investigative* features. In the event of attacks, it is extremely hard to tie the ends and come up with a thorough analysis of how the attack happened, what the steps were. In many cases, serious attackers are skillful at covering their tracks. Firewall logs and intrusion detection alerts are unlikely to be adequate for a serious investigation of a security incident, especially when the goal is to apprehend the perpetrator.

What is the solution? We believe the solution is in the realm of *Network Forensics* [3]; a dedicated investigation infrastructure that allows for the collection and analysis of network packets and events for investigative purposes. It is the network equivalent of a video camera in a local convenience store.

In this thesis, we are concerned with digital forensics in networked environments; i.e. *network forensics*. In some respects, network forensics represents all the complexities and challenges associated with moving from the forensic analysis of already complex standalone computers to the forensic analysis for networked computers. In particular, it has to deal with distributed computers and network devices.

A key challenge in network forensics is to ensure that the network itself is forensically-ready. In particular, a network must have an infrastructure to collect and analyze data in real-time. Such infrastructure is intended

to augment an existing network. It may employ software technology, hardware technology or both. Because of networks' distributed nature, designing such infrastructure is a challenging task. One, for instance, has to decide what/where data should be collected?

Another challenge arises because of attackers ability to move around in the network. In particular, it is common for a computer attack to originate from an attacker's computer, propagate through other computers, then attack a victim computer. This ability to move around in the network results in creating a chain of connections; commonly known as *connection chains*. They are widely used by attackers to stay anonymous and/or to confuse the forensic process.

Detecting and analyzing connection chains is a challenging, yet important task. Particularly, it has the potential of revealing an attack's path as well as the involved hosts. Investigation then typically proceeds by isolating affected hosts and collecting data from them. Ideally, this also may lead to the origin of an attacker especially insiders. Coupled with collected evidences, the attacker may also be prosecuted in a court of law.

Our goal in this thesis is to address the above challenges by developing an agent-based architecture for network forensics analysis with a focus on connection chains detection and profiling. We summarize in the next section the contributions made in this regard.

1.3 Contributions

Three main contributions are made in this thesis.

First, we propose an agent-based network forensics infrastructure, which is intended to add real-time network forensics capabilities into a controlled network. The proposed architecture employ different agents to support the distributed nature of computer networks and to separate the collection, logging and analysis processes.

Secondly, we propose a novel framework to detect connection chains. We adopt a black-box approach by passively monitoring inbound and outbound packets at a host, and analyzing the observed packets using association rule mining. We assess the proposed framework using public network traces, and demonstrate both its efficiency and detection capabilities.

Thirdly, we propose a profiling-based framework to investigate connection chains that are distributed over several ip addresses. The framework utilizes a simple yet extensible hacker model that integrates information about a hacker's linguistic, operating system and time of activity. We establish the effectiveness of the proposed approach through several simulations and an evaluation with a real attack data.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In chapter 2, we review various aspects in the field of network forensics and highlight related technologies.

We also present notions and relevant challenges. In chapter 3, we detail the architecture, implementation, and evaluation of the proposed network forensics system. In chapter 4, we present an extensive review and taxonomy of approaches for detecting connection chains. In chapter 5, we present a framework to detect connection chains. In chapter 6, we present a framework to profile connection chains. Finally, in chapter 7, we conclude the thesis.

Chapter 2

Network Forensics: Notions and Challenges

Network forensics is a young member of the bigger family of digital forensics discipline [4]. In particular, it refers to digital forensics in networked environments. In this chapter, we review various aspects in the field of network forensics and highlight related technologies.

2.1 Background

The term network forensics has been used in different contexts without an official definition [4]. However, it generally refers to the analysis of data collected on active networks devices, such as firewall logs, network traffic, IDS logs, etc. Essentially, this analysis is for investigating a security breach,

such as a system compromise.

Back in 1997, security expert Marcus Ranum coined the term *network forensics* [3], and described it as follows:

> *"For lack of a better term, we've been referring to this as "network forensics" instead of mapping blood patterns and DNA samples, we're looking at how to analyze packet traces and network connectivity graphs. Just like forensic scientists at a crime scene, we'd rather be able to re-construct events using a closed-circuit video camera's recording, than to have to use scuff-marks on the floor. Especially since network "scuff marks" are much, much easier to conceal."*

Technically, network forensics is a member of an already-existing and expanding field of *digital forensics* [4, 5]. In real life, forensic science refers to the use of proved scientific methodologies to answer questions related to criminal and civil litigation. Analogously, digital forensic refers to

> *"The use of scientifically derived and proved methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations [4]."*

In theory, digital forensics (and hence network forensics) is not a protection product. In particular, it is not supposed to replace firewalls and intrusion detection systems. Instead, it is a *process* in which tools, techniques, and human effort combine for the purpose of investigation.

Generally, digital forensics is of a great interest and importance. Law enforcement needs digital forensics in investigating crimes in which a computer or digital system is either being the target of a crime or being used as a tool in carrying a crime. In civilian applications, digital forensics is important in investigating anomalous activities and ensure service availability. For instance, the outcome of digital forensic can be used to set the filter rules of firewalls and update intrusion signatures in IDS systems.

2.2 Definition and Model

The following definition is adopted by the *Digital Forensic Research Workshop* (DFRWS) [4]:

> *"Network Forensics: The use of scientifically proved techniques to collect, fuse, identify, examine, correlate, analyze, and document digital evidence from multiple, actively processing and transmitting digital sources for the purpose of uncovering facts related to the planned intent, or measured success of unauthorized activities meant to disrupt, corrupt, and or compromise system components as well as providing information to assist in response*

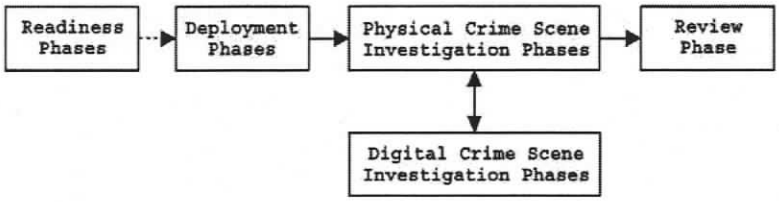

Figure 2.1: The Integrated Digital Investigation Process (IDIP)

to or recovery from these activities."

In general, digital forensic is a complex process. Accordingly, this also applies to network forensic. In the literature, few models have been proposed to model the digital forensics process [4, 5, 6, 7, 8]. There is no consensus about which model best (or even correctly) represents the process. However, the proposed models share a common foundation when fine details are ignored. In particular, they are based on standard investigation models that are applied in real-life crimes.

The *Integrated Digital Investigation Process* (IDIP) is a representative model of the digital forensic process [8]. It consists of various phases that are organized into five groups as shown in Figure 2.1. The following is a brief description of these groups:

- **Readiness phases**: The goal of these phases is to ensure that the personal and infrastructure are able to fully support an investigation when an incident occurs.

- **Deployment phases**: The goal of these phases is to provide a mechanism for an incident to be detected and confirmed.

- **Physical Crime Scene Investigation phases** : The goal of these phases is to collect and analyze the physical evidence and reconstruct the actions that took place during the incident.

- **Digital Crime Scene Investigation phases** : The goal of these phases is to analyze digital devices that were obtained from the physical investigation phases.

- **Review phases** : The goal of these phases is to review the whole investigation and identify areas of improvement.

2.3 State of the Art

Network forensics is generally a manual and brute-force process [9]. It is typically conducted by experienced system administrators. In some respects, it is arguably more art than science.

A typical network forensic investigation proceeds by analysing various types of logs. In a typical network setting, logs can be found in a number of places. For instance, a network is usually equipped with a dedicated auditing facility, such as *Syslogd* in Unix networks. Also, applications like web servers and network devices like routers and firewalls, maintain their own logs.

Various tools and homemade scripts are typically used for the investigation. For example in a Unix environment, an investigator may use free utilities like *tcpdump* [10], *grep*, *strings*, etc. Some investigators employ costly

commercial tools known as *network forensic analysis tools* [11, 12, 13]. The architectures of these commercial tools are not disclosed. However, they provide functionalities similar to those free utilities. Although, they are generally more user-friendly and versatile.

Since network forensics is generally a manual and brute-force process, it is usually both time consuming and error-prone. Additionally, the mentioned logs are not meant for thorough investigation. The logs may lack enough details or contrarily have lots of unrelated details. They also come in different incompatible formats and levels of abstraction. As such, there is a constant need for tool support.

2.4 Related Technologies

In this section, we review some relevant fields showing their connection with network forensics and their limitations.

2.4.1 Intrusion Detection Systems

overview

An *intrusion detection system (IDS)* is a system that monitors computing resources (a single host or an entire network) in order to *detect* attacks and security violations. In principle, it has the same goal as a "burglar alarm" installed in a house to detect trespassers.

To achieve their mission, IDSs collect various kinds of data from different

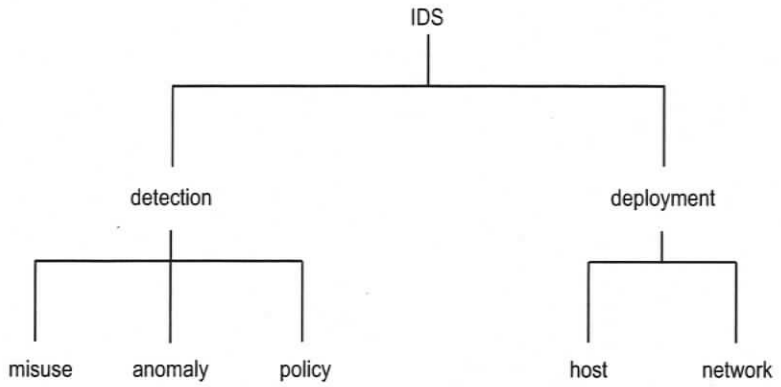

Figure 2.2: Classification of intrusion detection system (IDS).

sources and then process them employing a number of different approaches. Collectively, they can be classified according to their *deployment* and *detection* as shown in Figure 2.2. Firstly, in terms of deployment, an IDS can be either *host-based* or *network-based*. Whereas in terms of detection, there are three main approaches:

- *Misuse-based*: An approach where detection is achieved by *matching* against a database of known attacks. This approach is very similar to anti-virus software.

- *Anomaly-based*: In general, malicious activities should be different from normal ones. In this approach, an IDS builds a model of "normal" activities of a system. It then alerts when a deviation is detected.

- *Policy-based*: In this approach, usage policy of a system is explicitly stated. Accordingly, an IDS detects any policy violations.

network forensics

Network forensics is fundamentally different from IDS. First, it requires collecting far more data than what a typical IDS requires. Also, it is required to generate detailed output compared to an IDS's high-level alerts.

Within a network forensics framework, however, we believe that an IDS can be utilized for two purposes:

- Detecting an Attack: An IDS is used as a *sensor* that detects attacks, and subsequently triggers the forensics process. This is especially important for real-time systems.

- Profiling an Attacker: A class of anomaly-based IDS operates by *profiling* each user of a computer system [14, 15, 16, 17]. Theoretically, users can be distinguished from one another based on their patterns of usage of the computer system. Therefore, a user can be *profiled* by taking measures of his *"normal"* usage of the system. For instance, NIDES [14] use many measures like CPU usage, file usage, etc.

  If the profiles are reliable enough, they can be actually used to identify the users just like identification IDs. Ideally, one would build a profile "X" of an attacker while the attack is taking place. Then, the profile "X" is matched against a database of known users' profiles.

limitations

There are a number of issues pertaining to using IDS for network forensics. We summarize them as follows.

- Detection Reliability: When relying on an IDS's output, there are a number of concerns. First, IDS suffers from false alarms; namely *false-positives* and *false-negatives*. A false-positive refers to the case when an IDS generates an alert for a nonexistent attack. To the contrary, a false-negative refers to the case when an IDS misses an actual attack. In general, misuse-based and policy-based IDS have low false-positives and high false-negatives. Anomaly-based IDS are quite the opposite.

  A second concern is specific for network-based IDS. They can be a target for known classes of attacks; namely *evasion* and *insertion* attacks [18]. Also, they can not handle *encrypted* traffic.

- Data Details: In general, IDS's output lacks enough details for serious investigation. Typically, the output is a one-line text alert.

2.4.2 Honeypots

overview

The concept of a honeypot is simple, yet very powerful. It is a set of services, an entire operating system or even an entire network that is built to lure and contain intruders [19]. Although, honeypots are meant to be compromised,

they are in reality a tightly sealed compartment that is very well controlled and heavily monitored. An intruder will not have any idea that he is being watched.

A more precise definition of honeypots [20] is

> *A honeypot is a security resource, whose value lies in being probed, attacked, or compromised.*

It is a versatile tool that can serve different purposes. For instance, it can be setup to draw attention away from other real targets by directing malicious activities into it [21].

Essentially, all honeypots share the same concept. They do not have any production value or any authorized activity. Thus, any attempt to interact with them is most likely malicious.

network forensics

From an investigative perspective, a honeypot is an ideal tool to closely study hackers and capture their tools, keystrokes, etc. Few studies have been proposed to adopt honeypots for forensics purposes [22, 23]. A notable example, however, is the *Honeynet Project*; a voluntary research organization dedicated to study the tools, tactics, and motives of the bad guys [20].

limitation

From a legal point of view, honeypots can be problematic for at least two reasons. First, a honeypot has no value. It is solely setup to be compromised

and attacked. Therefore compromising it does not incur any damages. In other words, how could one assesses a damage when the victim has no value!

Secondly, honeypots can be regarded as a borderline between keeping attackers out of a network and inviting them in [19]. Therefore, they may be challenged as an unfair *entrapment*.

2.4.3 Computer Forensics

overview

Computer forensics is the oldest member in the digital forensics family. Traditionally, it refers to the forensics analysis and thorough investigation of standalone computers found in crime scenes. In particular, it involves analyzing their data storage devices, like hard disks. Typically, an investigator uses specialized software to recover deleted files, decrypt encrypted files, crack passwords, investigate found contact lists, etc.

Computer forensics has evolved over time following the standard methodologies used by law enforcement in investigating crimes in real life. Typically, the computer itself is not necessarily a victim of an attack, it is usually a tool used by a criminal. The forensics process usually follows well defined procedures to preserve, identify, extract, document, and interpret recovered data in the seized computer.

In general, computer forensics is not limited to personal computers. It also refers to investigating other digital devices that have some type of data

storage medium. Examples of such devices include cellular phones, PDAs, digital cameras, etc. Like computers, these devices can also be found in crime scenes or with suspects.

network forensics

When performing a network forensics investigation, computer forensics techniques can be employed to investigate the computers as if they were not networked. In other words, a networked computer can be isolated and analyzed as a standalone computer. Accordingly, computer and network forensics actually complement each other.

limitations

Computer forensics is solely for investigating standalone computers. It lacks in terms of investigating networked computers. In particular, it does not deal with issues that arise as a result of distributed sources of data. Such issues include data correlation, attack propagation, etc.

Additionally, computer forensics exclusively deals with persistent data stored on a local hard drive or other medium. In other words, the data is preserved when the computer is powered off. In networked environment, however, there is a need to deal with volatile data such as network traffic. Accordingly, the forensics process requires a *real-time* (or near-real-time) data collection and analysis.

2.5 Research Challenges

Network forensics represents all the complexities and challenges associated with moving from the forensic analysis of standalone computers to the forensic analysis of networked computers. In this section, we highlight some of the research challenges associated with network forensics. These challenges are then addressed in succeeding chapters.

2.5.1 Network Readiness

A key challenge in network forensics is to first ensure that the network is forensically ready. For a successful network investigation, the network itself must be equipped with an infrastructure to fully support this investigation [5, 4, 6, 7, 8]. The infrastructure should ensure that the needed data exists for a full investigation. After all, no investigation can be performed if data does not exist.

Designing a network forensic infrastructure is a challenging task because of the many possibilities in this design space. We will discuss several challenges when we introduce the proposed network forensic system in chapter 3. At a high-level, however, the challenges are briefly:

- Data Collection: In network forensics, data has to be collected in real-time. The challenge is to decide what/where data should be collected in the network.

- Data Analysis: The second major challenge is analyzing the collected

data, in order to produce useful information that can be used in a decision making process. Such analysis process is in many respect challenging due to the complexity of a typical network environment, the amount and diversity of data involved, the complexity and diversity of the attacks' tools and methods used. As a result, network forensic data analysis is a multifaceted problem with many different unsolved and partially solved issues. The analysis of connection chains in network forensics data is one such issue, which we discuss in the next section. Other issues include date and time synchronization of the collected data, dealing with emerging technologies (wireless for instance), managing legal and privacy issues, etc.

2.5.2 Connection Chains

Computer networks give attackers the opportunity to attack their victims indirectly. In particular, it is common for a computer attack to originate from an attacker's computer, propagate through other intermediary computer(s), then attack a victim computer. This leads to what is called *connection chains* [24]. Attackers resort to connection chains to hide the origin of their attacks and/or to confuse the network forensic process.

A *connection chain* is created when someone recursively logs into a host, then from there logs into another host, and so on as shown in Figure 2.3. Due to the design of TCP/IP suite, the origin of the chain is effectively concealed as we move down the chain. As such, a connection chain actu-

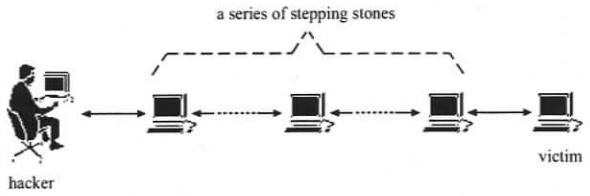

Figure 2.3: Using a connection-chain to hide an attacker's origin.

ally provides an interactive channel to remotely manipulate a host without revealing someone's origin.

Connection chains can be further complicated to confuse the network forensics process. In Figure 2.4, two variations of connection chains are shown. In the first scenario, two independent attackers use the same intermediary host to attack a victim machine. In this case, a forensic process would mistakenly aggregate/correlate data from two different attackers and hence two different attacks.

In the second scenario, an attacker uses two different intermediary hosts for his attack. The victim will see two different ip addresses and has no way to tell that they are originating from the same hacker. Consequently, data and alerts from the two ip addresses would pass uncorrelated.

Technically, detecting and analyzing connection chains belongs to the *data analysis* class of challenges that were mentioned earlier under network readiness. In chapters 4, 5 and 6, we study them in greater details and propose methodologies to address them.

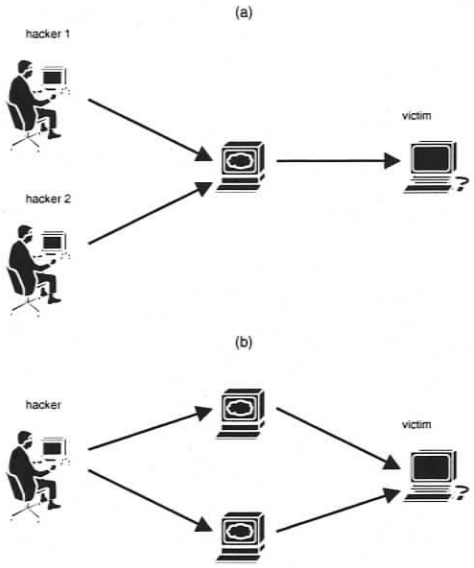

Figure 2.4: Two scenarios which confuse a network forensic process. (a) Two independent hackers use same intermediary node to attack a victim. (b) One hacker use two intermediary nodes to attack the same victim.

Chapter 3

Agent-Based Network Forensics System

For a successful network investigation, the network itself must be equipped with an infrastructure to fully support this investigation [5, 4, 6, 7, 8]. The infrastructure should ensure that the needed data exists for a full investigation. After all, no investigation can be performed if data does not exist.

In this chapter, we detail the architecture, implementation, and evaluation of a proposed network forensics system. The system is intended to add real-time network forensics capabilities into a controlled network. The proposed architecture employs different agents to support the distributed nature of computer networks and to separate the collection, logging and analysis processes.

Several concepts are employed in this design. In particular, the proposed

system uses:

- A network-based *watch list* to maintain a list of offending ip addresses.
- A network-based marking scheme to distinguish packets originating from offending ip addresses.
- The ability to support several monitors (IDS for instance) to improve detection efficiency of the system.
- A network-based logging facility.
- Host-based agents to collect data at hosts in order to circumvent encryption.
- A set of lightweight, intelligent and specialized analysis agents, which help to answer some of the main questions raised during a typical network forensic analysis process.

3.1 Design Space

Designing a network forensics system poses a number of challenges which includes the following:

- *Data Capture*: The system has to capture data in real-time, which raises a number of issues:
  - Sources of Data: where should the data be captured? This leads to three possible locations: host, network or both.

  - Details of Data: How much data should be captured? There are many possibilities depending on where the data is captured. At network level, we may capture entire raw packets or just summaries like source/destination ip and port numbers, protocol, etc.
  - Integrity of Data: the system has to ensure the integrity of collected data.

- *Detection Efficiency*: The system has to detect attacks in order to trigger the forensics process. An IDS is a natural choice for this task. IDSs use different detection approaches; namely misuse-based, anomaly-based and policy-based. Each approach has its own advantage and disadvantages. To increase the system's detection efficiency, the system should accommodate for various types of detection techniques.

- *Data Analysis*: After collecting the data, it is important to correlate them together. In particular, we would like to be able to reconstruct an attack the way it happened.

- *Attacker Profiling*: The system has to maintain information about the attacker himself. For instance, it can identify the attacker's operating system through passive OS fingerprinting.

- *Privacy*: Depending on the application domain, privacy issues can be a major concern.

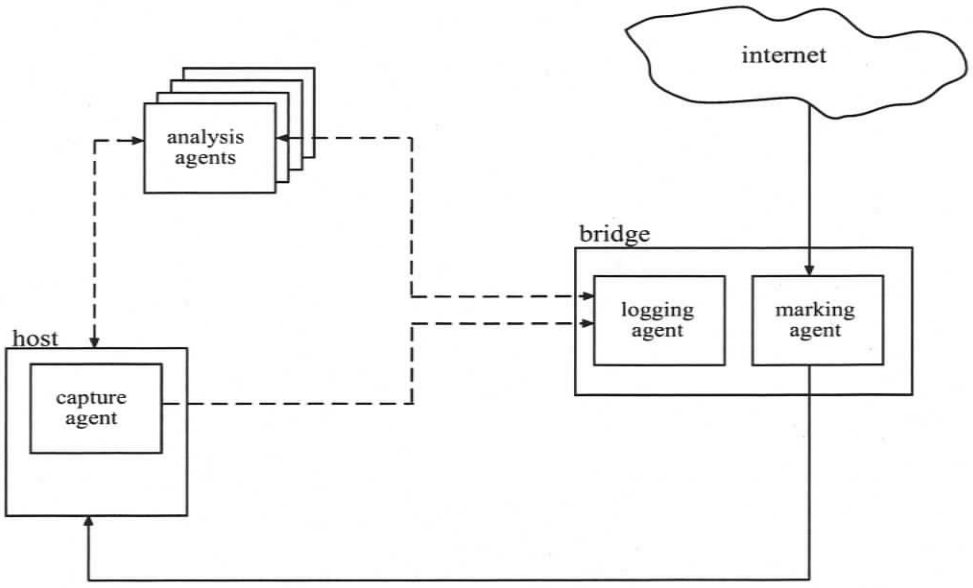

Figure 3.1: The overall architecture of the system

- *Data as Legal Evidences*: For the collected data to qualify as evidences in a court of law, they have to be collected and preserved correctly to pass the *admissibility* test; a screening process by the court [25]. They also have to be *authentic*, *complete*, *reliable* and *believable* [26].

3.2 The System Architecture

In Figure 3.1, the proposed architecture is depicted. The system consists of four main entities:

- *A Marking agent*: An agent to identify and mark malicious packets as they enter our network.

- *Capture agents*: agents installed on all the hosts to gather marked packets and post them to a logging facility (see next item).
- *A Logging agent*: A logging facility responsible for archiving the captured data.
- *Analysis agents*: agents to analyze the collected data.

In a typical network, those agents are placed as shown in Figure 3.1. The marking agent and the logging agent are network-based, while the capture agents are host-based. They together form a kind of closed circuit. An incoming packet first passes through the marking agent which marks "malicious" packets. Subsequently, when a host receives a marked packet, it posts it to the logging agent. At last, analysis agents are task-specific processes for analyzing the collected data. Each agent will now be explained in further details.

3.2.1 Marking Agent

This agent is the entry point to our system on which subsequent agents depend. It is in charge of the difficult task of deciding whether a passing-by packet should be considered friendly or malicious. Then, it marks the packet accordingly. Figure 3.2 depicts the architecture of this agent. It consists of three main components:

- Sensors: One or more network-based sensors to report suspicious ip addresses to the *watch list*. Technically, a sensor is a software that

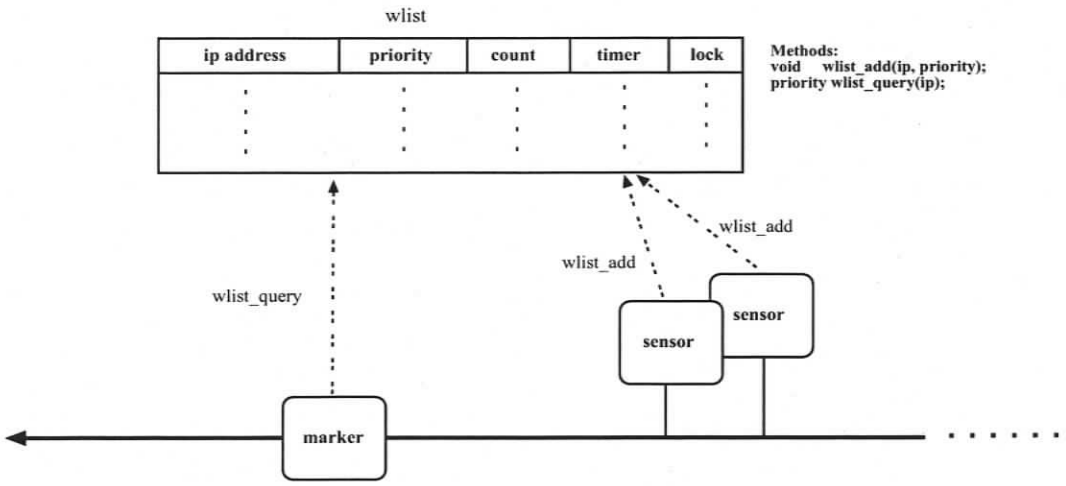

Figure 3.2: The marking Agent

sniffs traffic, detects suspicious actions, and reports the corresponding ip addresses. Two examples of sensors are

- An intrusion detection system (IDS) that maintains a database of known attacks. If a packet matches a known attack the corresponding ip address is reported. As the system evolves, this database of attacks is updated with new attacks.
- A policy-based sensor that enforce certain policies. For instance, a policy may state that a machine hosting a web server should not initiate an ftp or telnet connection. Detecting such an action is a violation of the policy, and the concerned sensor would then report the incident.

Allowing the use of diverse sensors is essential to increase the system's

detection efficiency.

- A Watch List (wlist): A list of bad ip addresses. It will be discussed in more details shortly.

- A Marker: A process to mark packets. Before sending a packet to its way, it queries the watch list to check whether the packet's ip address is in the list. It accordingly modifies the *type of service field (TOS)* in the ip header. The TOS field is an 8-bit field which is generally unused; i.e. it is typically set to 0. Therefore, we use the TOS field to differentiate those packets coming from offensive ip addresses, in order for the receiving hosts to recognize them. Specifically, the TOS field is set to one of three values that correspond to the offense levels: HIGH, MEDIUM or LOW.

The *watch list (wlist)* is a data structure which maintains a list of the current system's offenders. One may think of it as a cache memory of bad ip addresses. Each row corresponds to a unique ip address reported by at least one of the sensors. For every bad ip address, the list also maintains the following information:

- *priority*: It stores a *priority* measure, which indicates the current offense level of this ip address. Three levels are defined: HIGH, MEDIUM and LOW. A sensor must be able to classify an attack into one of these three levels. Also, this field always contains the highest level reported.

In other words, if different priorities for a single ip address were reported, the list only keeps the highest.

- *count*: It is a counter which is incremented every time the corresponding ip address is reported.

- *timer*: It is a count-down timer which is decremented automatically every second. If it reaches zero, the corresponding row is removed from the list. This field is set to a certain value when the ip address is first added. It is also reset to that value every time the ip address is reported. One may think of this field as a sliding time window. If an ip address was not seen for a long time (say 1 week), we may remove it from the list.

- *lock*: This field is to synchronize accesses. It is needed because the list is asynchronously accessed by a number of processes.

To interact with *wlist*, two methods are provided:

- *wlist_add(ip, priority)*: A method to add an attacker's ip address populating the watch list. In addition to the ip address, it requires a priority measure: HIGH (1), MEDIUM (2) or LOW (3).

- *wlist_query(ip)*: A method which returns the priority of a given ip address if it exists in the list.

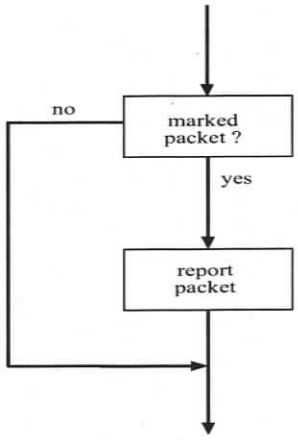

Figure 3.3: The Capture Agent.

3.2.2 Capture Agent

The second major component in our architecture is a collection of capture agents. They are lightweight agents, which reside silently in the hosts waiting for marked packets. They then arrange to reliably transport them to the logging agent for safe archival. This transportation is necessary because we cannot store the data locally. Once a system has been compromised, it cannot be trusted.

The capture agent is host-based and hence implementation-specific. Unfortunately, it is very hard to come up with an architecture that fits several operating systems or even different version of the same operating system. However, in general, it should provide the operation shown in Figure 3.3.

Installing capture agents in the host is essential for two reasons. First, attackers increasingly use *encryption* to hide their actions, rendering traffic

sniffing useless. Even if we use trojaned encrypted service (like secure shell), attackers usually use their own encrypted channels. Therefore, only at the host, we can circumvent encryption and fully record an attacker's actions.

The second reason concerns the attack itself. There is no guarantee that a malicious packet will actually compromise or damage a host. In fact, the packet may be directed to a nonexistent service or even a nonexistent host.

3.2.3 Logging Agent

The logging agent is our system's repository where attack data are being stored. Ideally, one would turn to this agent for reliable answers and documentation about any attack.

In general, most network systems are supplemented by a logging facility. For example, in a typical *Unix* environment, *syslogd* is a standard logging daemon that reads and logs messages to the system console or log files. The message sent to syslogd is restricted, however, to only a single line text message. It is a useful facility but logging for network forensics purposes clearly requires more than simple one-line text messages.

When designing a logging facility, there are a number of problems we need to address. First, we need to determine the appropriate sources of data that are relevant. Understandably, there are several possible candidates. Also, a related problem is how much data can safely be considered enough. Specifically, what level of details one should include. More details mean more information about the situation. It, however, requires more storage

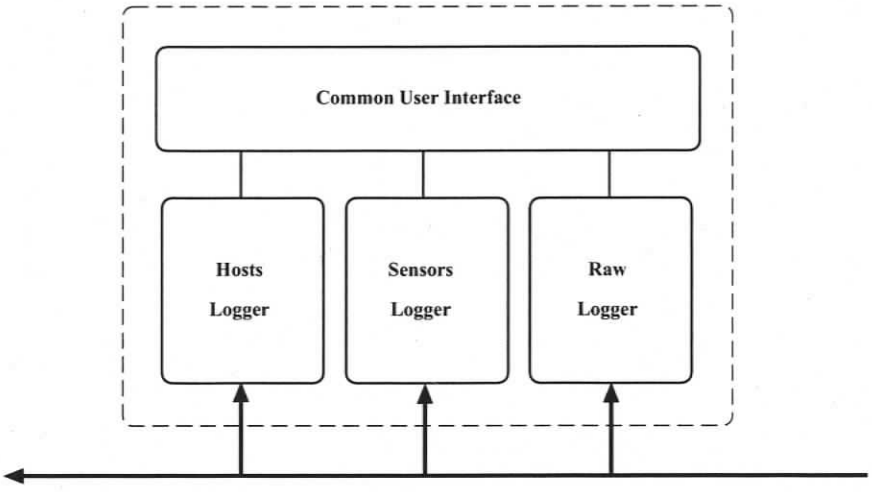

Figure 3.4: The Logging Agent.

and more time to process. On the other hand, less details saves us some storage space and processing time. But, it may lead to missing important details. Therefore, we need to aim for a trade-off that provides a reasonable balance between level of details, storage requirement and time requirement.

In Figure 3.4, we show the proposed architecture of a network-based logging agent. We propose to collect data from three different sources using three loggers.

- Hosts Logger: This logger is responsible for logging data gathered by the capture agents installed in the hosts. It is expected to log detailed data pertaining to real attacks. Both time and storage overheads are not high though. After all, attacking a host should not be happening all the time.

- Sensors Logger: This logger stores the sensors' alerts. Although, a typical alert is only a one-line text message, it provides a quick diagnosis about an attack. Both time and storage requirements here are low.

- Raw Logger: This is an optional logger which provides a last resort solution when other loggers fail. It archives raw packets straight off the line. In busy networks, however, this logger is expected to require an excessive amount of storage.

The last part in this agent's architecture is a layer that should provide a common user interface to access these loggers.

3.2.4 Analysis Agents

The fourth major component in our architecture is a set of analysis agents. They are intended to support the analysis of the collected data. In particular, they are supposed to consume raw data and produce useful information for an investigator.

An analysis agent is designed to perform a specific task. Ideally, it should provide specific and simple answers to specific queries. In forensic science, a DNA test (for instance) can be used to answer questions about an individual identity. Analogously, an analysis agent should be designed with similar perspective.

An analysis agent can be simple or complex, may have straightforward goals or deal with more challenging tasks. For instance, passive fingerprinting

tools can be used to analyze the collected data and identify the operating system of the attack machine. Another instance of analysis agent, which is currently being investigated in our research lab, consists of biometric-based profiling [27]. The goal is to establish the identity of the attacker by deriving a biometric signature (fingerprint) from his keystrokes and mouse dynamics while interacting with a victim computer.

Technically, an analysis agent is not restricted to certain methodology and/or architecture. Depending on it's task, it can employ analysis methodologies such as *data mining* [28], *clustering analysis* [29], etc. Similarly, it's architecture can be *distributed*, *modular*, etc.

Overall, we view analysis agents as lightweight, intelligent and specialized modules which help to answer some of the main questions raised during a typical network forensic analysis process. In particular, an agent is designed for a specific class of attacks or anomalies. In this thesis, we focus our effort on the study of an analysis agent that is designed to answer queries about connection chains at a host. As more attacks and anomalies are detected, we expect the system to evolve by introducing more specialized analysis agents.

3.3 Implementation

As a proof of concept, we built a prototype that implements the proposed architecture. The goal is to test our approach and draw some initial results. Specifically, we wanted to see if the system can reliably document attacks as

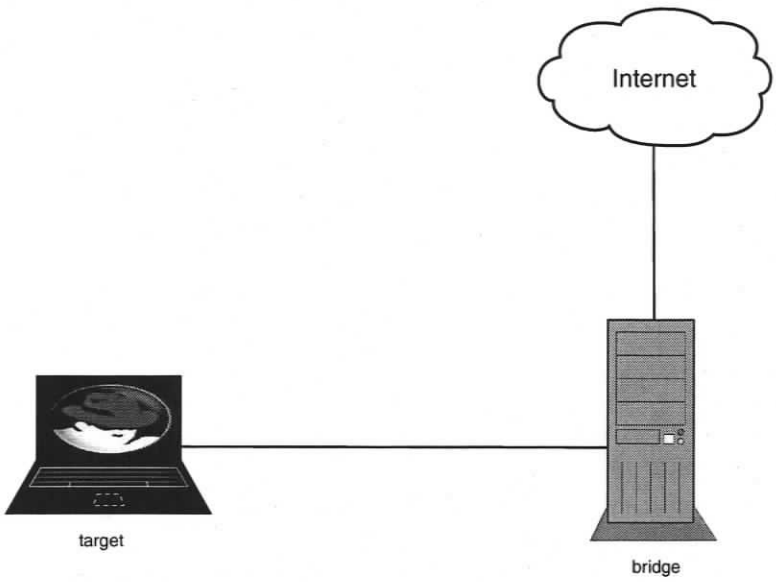

Figure 3.5: The Prototype

they happen.

In this section, we describe the implementation of the prototype. The prototype, implemented as depicted in Figure 3.5, consists of two PCs; namely a target (or victim) and a bridge. A capture agent is hosted in the target machine, while the marking and logging agents are hosted in the bridge. At this stage, we did not implement any analysis agents. Instead, all analysis were done manually. In Chapters 5 and 6, we describe in details two analysis agents for detecting and analyzing *connection chains*.

In this prototype, the experimental network consisted of one host; i.e. the target. The results would be similar, if we used more hosts. In that case, we simply need to install capture agents in each host. In section 3.4.3, we

further discuss scalability issues related to our design.

In the rest of this section, we describe in details the target and bridge in this prototype.

3.3.1 The Target

The first PC is a laptop with a 400MHz Pentium II processor, 132 MB RAM and 6GB hard drive. It was setup to be a potential target for attacks. It is fully accessible from the Internet via the bridge described below. It is physically attached to the bridge using an Ethernet cross-cable. To allow attacks in, we installed a relatively old Linux distribution; namely RedHat 7.1. Also, we enabled two vulnerable services: FTP (wu-ftpd 2.6.1-16), and RPC. We chose Linux because it is free and open-source. Using other operating systems would lead to same results and conclusions.

According to our design, we require a capture agent to be installed in the host. For this purpose, we installed *sebek* [30] – a tool from the *Honeynet* project [20]. Sebek is a kernel-based data capture tool which circumvent encryption by intercepting the *read* system call at the host.

3.3.2 The Bridge

The second player in our prototype is the bridge. It hosts both the marking and logging agents. It is a PC with a 1.7GHz Celeron processor, 512MB RAM, 40GB hard drive and 2 network cards. We installed a custom Linux

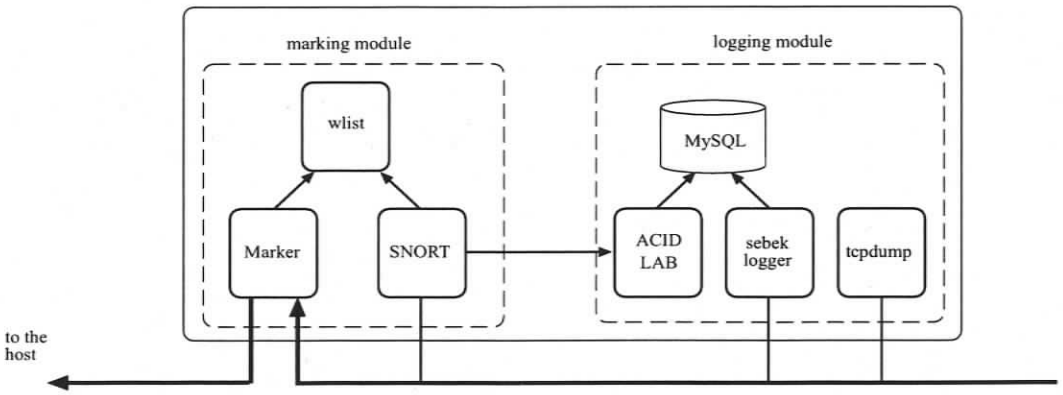

Figure 3.6: The Bridge Internal

operating system and a collection of tools and homemade programs which reflect the proposed architecture. Figure 3.6 shows the internal architecture.

The marking agent follows the architecture described earlier. One sensor was used; namely SNORT[1] [31]. Both the watch list *(wlist)* and the marker was implemented using C language and *pthread* – the multi-threading standard library. For each packet, the marker sets the *TOS* field in an IP header according to the returned priority.

The logging agent also follows the proposed architecture. It is comprised of three loggers and MySQL [32] as a backbone database. The first logger is to log packets captured by the host. Since *sebek* was used to capture packets at hosts, we used its corresponding server-side tools, which include a tool to log packets to MYSQL and a navigation web application front-end. The

[1]Snort is a free intrusion detection system (IDS) [31]. It belongs to the class of network-based misuse-based IDS. It can perform protocol analysis, content searching/matching of network traffic in real-time. Accordingly, it can be used to detect a variety of attacks and probes.

second logger is for the sensor: SNORT. We used the *barnyard* tool from SNORT project to log alerts to MYSQL and *ACID Lab* [33] to analyze the alerts. Finally, we chose to use *tcpdump* [10] as a raw packets logger just in case we miss something.

3.4 Experiment

3.4.1 Approach and Setup

We run an experiment over a period of 12 days from March 17^{th} until March 28^{th}, 2004. The prototype was connected to the Internet using a local ISP, making it accessible for our research group and strangers as well. The system's ip address was not advertised. It was, however, given only to our research group.

During the experiment, the system was compromised three times. Two of which were committed by total strangers from the wild. In all these cases, however, the attacks were detected and well documented making it possible to reconstruct an attack. Overall, the experiment proved to be extremely useful.

3.4.2 Results

General Statistics

In this section, we look into some general statistics about the traffic that was directed to the target host. Table 3.1 lists the number of packets directed to the target.

	invited packets	wild packets
TCP	70130	133216
UDP	8928	9581
ICMP	5150	6986
total	84208	149783
	36%	64%

Table 3.1: Number of invited and wild packets directed to the target by protocol type

	count	size
SNORT	3482 alerts	$111KB$
sebek	336132 packets	$38MB$
tcpdump	734500 packets	$69MB$

Table 3.2: Storage requirement for each logger

First column lists the protocol types of the packets. Second column is marked "invited packets". It lists the number of packets generated by our research group members who participated in this experiment. The last column is marked "wild packets". It shows the packets coming from uninvited strangers.

It is a surprise that a totally anonymous unimportant ip address receives so much scans knowing that it was neither advertised nor located in a popular organization or company. As shown in the table, 64% of the traffic was not invited. In fact, the target has been probed and actually compromised by

two of the uninvited strangers! In the next section, I will provide a detailed discussion about a complete compromise carried out by someone outside our research group. He did not only compromise the system. He actually installed a rootkit [2] enslaving the machine.

The table also shows a break down of the number of packets received based on the type of protocol. As shown, TCP is more prevailing than other protocols. For the wild traffic, TCP packets are about 10 times and 20 times more than UDP and ICMP packets respectively.

Table 3.2 sorts the storage requirement for the three used loggers in ascending order. SNORT requires the least amount, while tcpdump requires the most. Although, sebek is a powerful tool in honeypot settings, it actually did not fit our need. It captures far more data than we need. As the table shows, it requires a relatively excessive storage. In the future, we are planning on developing our own tool.

A Detailed Attack

In this section, we discuss one of the successful attacks that happened during the experiment. It was carried out by an uninvited stranger who found our vulnerable ftp server. He then successfully compromised the target machine and installed a rootkit. Figure 3.7 shows a time-line diagram of his steps.

As shown in the figure, he first found the ftp server (wu-ftpd 2.6.1-16).

[2] A rootkit is a malicious software used to hide running processes, files or system data in an operating system. It is used by hackers to maintain access to systems while avoiding detection. A rootkit typically modifies critical parts of the operating system. In Unix, for instance, a rootkit usually replaces standard utilities like *ps*, *netstat*, etc.

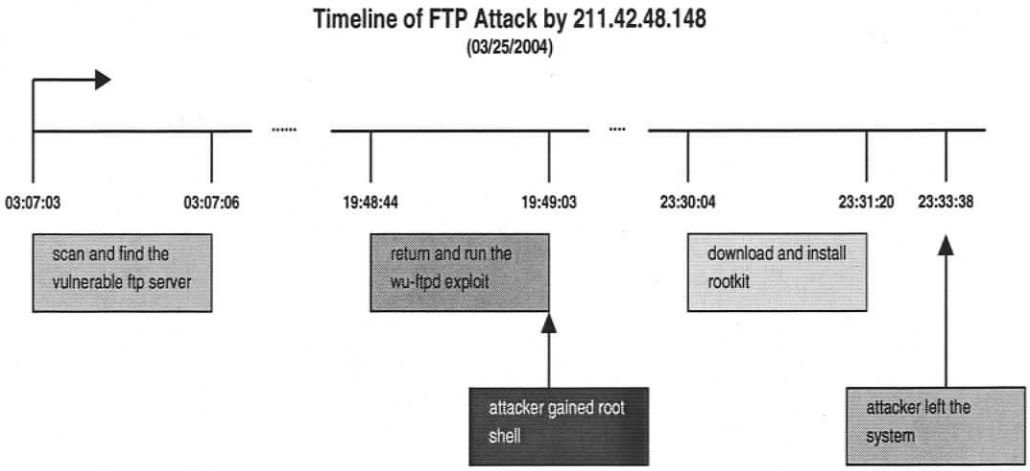

Figure 3.7: Time Analysis of one of the attacks on our ftp server

This ftp server has readily available exploits which can be easily searched for. Therefore, he returns back after about 16 hours with a workable exploit. He then runs the exploit and immediately gains a root shell. He left the connection open until later that night after about 4 hours.

Upon returning, he typed a number of commands and then exited. The following is a recreation of those commands.

```
[23:28:52] w
[23:29:54] wget
[23:30:04] wget 65.113.119.148/l1tere/l1tere.tgz
[23:30:19] ls
[23:30:24] tar xzvf l1tere.tgz
[23:31:20] ./setup
[23:33:38]
```

First, he runs the *w* command to check who is logged-on to the system.

Then, he checks if *wget* (a program for downloading) is installed; *wget* responds by complaining about missing arguments. This indicates that the program is actually installed. He then proceeded by downloading his rootkit from an IP address which points to a site for free web hosting. Now the rootkit is in place as he double-checks by running the *ls* command. Next is un-tarring the rootkit and starting the installation by running the *setup* script.

Figure 3.8 shows a snapshot of what the attacker console might look like. The first shot shows running the exploit which gives a root shell. Then, the commands are continued in the second shot.

It is interesting to note that the attacker first compromised the system then waited for about 4 hours probably wondering if nobody has noticed. Then he quickly downloaded and installed a rootkit then left in about 5 minutes. He actually showed up later after 2 days. However, we already took the system offline, reinstalled the operating system and disable the ftp server.

Before leaving this section, we would like to give a brief description about our analysis of the installed rootkit. A two step analysis was conducted. We first tested the compromised system itself. We were also curious to know whether this is a new rootkit or an already known one. We run *chkrootkit* [34] – a free tool that checks for signs of a rootkit. *chkrootkit* found a number of trojaned binaries and warned a rootkit named *showtee* may have been installed. This however may not be accurate for two reasons. First, a number

Figure 3.8: Attacker's console

of binaries on the system were actually trojaned but reported safe. We actually know this because we have downloaded the kit itself as discussed shortly. Also, warning about the *showtee* rootkit is based on a simple check whether certain files exist or not.

A more thorough analysis was done using a fresh copy downloaded from the address used by the attacker. The analysis was done by manually going through the various files enclosed in the kit. The following is an enumeration of the main impacts of this kit:

- creates directories and files under */lib/security/www/.*
- removes other rootkits.
- replace some binaries with trojaned ones; many to mention!
- shut down the RPC *portmapper* at port 111.
- installed a sniffer and a *SSHD* backdoor.
- updates the *init.d* scripts.
- disable the anonymous vulnerable ftp server.
- send an email to l1tere@yahoo.com with detailed information about the found treasure!
- cleans up and delete downloaded files.

Overall, this particular attacker generated about 1100 packets. He also caused a 158 *SNORT* alerts. Only two alerts had priority one meaning high risk. For the remaining 156 alerts, 154 were medium risk and 2 were low

risks. On another note, the passive fingerprinting tool *p0f* [35] indicated that he is most probably using a Linux 2.4/2.6 operating system.

Assessing the Results

Assessing the results are informal at this stage. We, however, can safely argue that we were able to detect and reconstruct all the compromises of the target.

The proof pertains to using *sebek* at the target. In particular, *sebek* (besides many things) captures keystrokes at the host and transport them to the corresponding logging facility. We setup the target in such a way that it should not be accessed. Therefore, seeing any keystrokes means a compromise!

Also, *SNORT* (our sensor) is aware of the relatively old vulnerable *ftp* and *RPC* servers. This gave us another indication of an ongoing attack.

3.4.3 Discussion

In this section, we discuss some of the issues related to the proposed architecture.

Reliability

When designing a network forensics system, one of the main concerns is to ensure that the system can reliably detect an on-going attack and collect data about it. The proposed architecture addresses this issue by allowing the use

of several sensors. A sensor can be an "off-the-shelf" intrusion detection system (IDS), or a specialized sensor that detects certain class of attacks or anomalies. We believe that this collaboration of several sensors would greatly improve the overall reliability of the system.

Performance and Scalability

Considering the proposed architecture, the marking agent is apparently the bottleneck in the system. Therefore, it is expected to be the main concern in terms of performance and scalability.

In our experiment, we evaluated the performance overhead caused by running the marking agent in the bridge as follows. We used a tool called *bing* [3] [36] to measure the bandwidth (in bps) of the bridge with and without the marking agent running. Our measurements indicated that the bandwidth has actually dropped when running the marking agent. The reduction, however, is less than %1.

Although the overhead is reasonable, it is possible to improve the performance and scalability of the system by distributing the marking agent over several machines. For instance, we may host every sensor in a separate machine.

[3]Bing is a point-to-point bandwidth measurement tool. It determines the real bandwidth on a link by measuring ICMP echo requests round-trip times for different packet sizes for each end of the link.

Data Analysis

In our experimentations, all data analysis was performed manually similar to the current practice. We experienced, firsthand, the data analysis process. It is a time consuming and error-prone process. It also requires diverse skills and expertise to link various clues found in the data.

We believe that the key issue in any network forensic infrastructure is data analysis. In the proposed architecture, this key issue is addressed by a set of specialized analysis agents. Instead of dealing with all attacks in a monolithic way, we propose the use of specialized analysis agents that are able to answer specific queries.

3.5 Conclusion

A network forensics system can prove to be a valuable investigative tool to cope with the attacks that computer systems have to deal with these days. If well designed, they can also provide legal binding evidences of abuse and malicious activities.

In this chapter, we proposed an architecture of network forensics system, and then discussed our implementation of the proposed architecture. We also deployed and studied the system in a real-life environment. The proposed system manages to collect data related to attacks unlike the brute force manual approach, where data is scattered everywhere for an investigator to look for. Also, an investigator does not have to sort through unrelated

data; all data is attack related. Finally, the system is capable of capturing encrypted activities at the hosts. This is an advantage over automated tools which cannot handle encrypted traffic, for instance to capture an attacker's keystrokes.

One of the key aspects of the proposed architecture is implemented by analysis agents. We view analysis agents as lightweight, intelligent and specialized modules which help to answer some of the main questions raised during a typical network forensic analysis process. One such issue is the analysis of connection chains which raises several challenges. In the next chapter, we survey existing work on connection chains and elaborate a related taxonomy. Then, in chapters 5 and 6, we introduce two different complementing analysis agents which address the issues of connection chains detection and profiling, respectively.

Chapter 4

Connection-Chains: Review and Taxonomy

A *connection-chain* is a set of connections created by sequentially logging into a series of hosts, known as *stepping-stones*. It provides an effective scheme for attackers to manually interact with a victim machine without disclosing their true origin. The victim will only identify the last host in the chain, while the true origin is hidden behind a series of stepping-stones. Addressing connection-chains poses challenges for researchers in the field of computer security. Accordingly, several approaches have been proposed in the literature. In this chapter, we review those approaches and classify them according to a proposed taxonomy.

4.1 Introduction

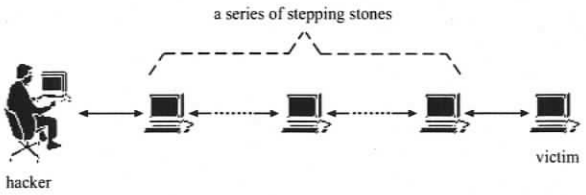

Figure 4.1: Using a connection-chain to hide an attacker's origin.

The increase in hacking activities over the Internet is attributed to a number of factors. Probably, one important factor is the lack of *accountability*. Attackers have plenty of tricks and techniques that help them to stay *anonymous* during their attacks.

A very effective anonymity technique is to indirectly attack a victim machine via a series of intermediary hosts; a scheme that is often called a *connection-chain*. The chain is established by recursively logging into different hosts (known as *stepping-stones*) before attacking the target machine as shown in Figure 4.1. Effectively, the connection-chain constitutes a channel that connects the attacker on one side with a victim machine on the other side. It gives the attacker an anonymous mean to manually interact with the victim machine without revealing the attacker's origin. The victim will only see packets coming from the last host, when in fact; the attack is hidden behind a list of possibly unrelated hosts.

Tracing connection-chains is a challenging yet important task for a number of applications. The following is a sample:

- Network Forensics: Tracing connection-chains plays a crucial role in network forensics applications. Particularly, it has the potential of revealing an attack's path as well as the involved hosts. Investigation then typically proceeds by isolating affected hosts and collecting data from them. Ideally, such tracing also may lead to the origin of an attacker especially insiders. Coupled with collected evidences, the attacker may also be prosecuted in a court of law.

- Liability: If a host owned by an organization were exploited as a stepping-stone, the attack would appear to be originating from this organization. As a result, this organization may be held liable for such attack. Detecting connection-chains can help to enforce policies of transit traffic.

- Deterrence: Anonymity is a main concern of serious attackers. In fact, it is the whole purpose of establishing a connection-chain in the first place. An effective tracing tool will deter some attackers in fear of exposing their true origin.

Historically, connection-chains have been used repeatedly by attackers to hide their true origin. For instance, they have been used in a spy chase documented in the popular book: *the cuckoo's egg* [37]. Yet, Staniford-Chen and Heberlein are first to actually address the problem within a network security context and propose a solution [24]. They also coined the term *connection-chain.* The term *stepping-stones* was later coined in [38]. Since

then, different approaches for detecting and tracing connection-chains have been proposed in the computer security literature.

In this review, we survey several approaches for detecting and tracing connection-chains. We also classify these approaches according to a proposed taxonomy. The review focuses on the *technical* issues. Specifically, we try to show how an approach works and highlight some shortcomings as well. There are some *non-technical* issues, for instance legal and societal ones. Such issues are not discussed here, but the interested reader is advised to reference the paper by [39].

The rest of the review is outlined as follows. In section 4.2, we address some related and subtle issues surrounding connection-chains. In section 4.3, relevant terminology and background material are presented. Then, taxonomy of connection-chains approaches is briefly described in section 4.4. The taxonomy is then explored in further details in the succeeding sections. First, network-based approaches are discussed in section 4.5. Then, host-based approaches are discussed in section 4.6. At last, system-based approaches are discussed in section 4.7. In section 4.8, we evaluate the reviewed approaches against a set of criteria. Finally, we conclude the review and present possible open challenges in section 4.9.

4.2 Related Issues

4.2.1 Anonymity: Connection-Chains Vs Spoofing

Serious attackers may have different motives, but definitely share a common concern; i.e. *anonymity*. They strive to hide their true origin during their attacks by employing different tricks and techniques. Generally, these techniques belong to one of two major schemes. One of them is using connection-chains, which is the focus of this review. The other one is to carry an attack using spoofed traffic; i.e. traffic where packets have forged source `ip` addresses.

From a tracing perspective, the settings of the two schemes are very different. As a result, researchers have treated them almost in parallel. *IP traceback* or just *traceback* [1] is coined for tracing spoofed traffic, while plain terms like *tracing* or *tracking* are used in the context of connection-chains. In what follows, we highlight some of the main differences between the two schemes.

- A connection-chain is established to create a bidirectional channel for an attacker to manually interact with a victim machine. As a result, every connection in the chain must have correct source and destination `ip` addresses. On the other hand, spoofed traffic is used for flooding denial of service attacks (DOS/DDOS). An attacker is not interested in receiving any traffic from the victim machine. Therefore, source `ip`

[1] For *IP traceback* approaches, the interested reader is referred to a review by [40]

addresses can be anything.

- In connection-chains, anonymity is achieved because a victim can only trace traffic to the last host in the chain. The true origin is laundered by several intermediary hosts. In spoofing attacks, however, an attacker stays anonymous because the packets' source `ip` addresses are fictitious.

- The type of traffic underlying the two schemes is very different. Connection-chains carry interactive traffic that reflects an attacker's typing dynamics. Therefore, the packets typically carry few (mostly one) bytes with timings of a typewriting human. On the other hand, spoofed traffic is generally massive and only limited by the dynamics of the network.

4.2.2 Legitimate Connection-Chains

Many users establish connection-chains on a daily basis either for convenience or necessity. A typical example is accessing restricted machines. These types of connection-chains are legitimate, because they conform to typical usage policies. The focus of this review, however, is the "bad" type of connection-chains. Distinguishing one from another is a subtle issue that may cause some confusion. In this section, we try to highlight what makes a connection-chain bad.

One distinguishing criterion is the *objective* of creating a connection-chain in the first place. As mentioned earlier, legitimate connection-chains are established for either convenience or necessity. On the other hand, illegitimate

ones are established by someone to hide her true origin; i.e. to stay anonymous. Clearly, the objective is very different in the two cases. Unfortunately, there is no easy way to identify the objective of a connection-chain. However, the next criterion can help.

Another distinguishing criterion is the association of a connection-chain with an attack. A connection-chain by itself is not an attack. In fact, it is just a channel to carry an attack. Therefore, legitimate connection-chains should not carry any attacks. On the other hand, illegitimate ones should carry signs or signatures of some attacks. Accordingly, "bad" connection-chains are those that can be associated with an attack.

4.2.3 Progressive Difficulty

From a computer security perspective, connection-chains are troublesome since they are quite easy to establish and use for attacks. At the same time, they are hard to trace as they may span different autonomous systems (AS). Additionally, neither the standard TCP/IP suite nor standard operating systems adapt protocols or mechanisms to deal with them.

The difficulty level of tracing a connection-chain is related to its environment. Specifically, the difficulty is inversely proportional to the ability of controlling the hosts and the network. This progressive difficulty can be demonstrated with the following three reference models [41].

1. *Closed Model*: Both hosts and network are under the control of a central

authority.

2. *Academic Model*: A central authority controls the network, but not the hosts.

3. *Internet Model*: Neither the hosts nor the network are controlled by a central authority.

In reference to the above models, the challenges in detecting and tracing connection-chains progressively increase as one moves from one model to the next one.

4.3 Background Knowledge

4.3.1 Terminology and Definitions

In TCP/IP suite, applications like `telnet` [42], `rlogin` [43] and `ssh` [44] are used to log in a host and acquire a *virtual terminal* (or simply a *terminal*) on that host. The terminal (also called *console* or *shell*) is useful to execute commands and other programs *interactively*. For convenience, we refer to such applications as *terminal applications*.

If a user runs a terminal application on host h_0 to log into another host h_1, a terminal on host h_1 is obtained and a TCP *connection* [45] (or simply a *connection*) c_0 is established. The user then may use the terminal at host h_1 to log into another host h_2. This procedure may be repeated as many times

as the user wishes [2] creating a series of connections as follows:

$$|h_0| \longleftarrow c_0 \longrightarrow |h_1| \longleftarrow \cdots\cdots \longrightarrow |h_{n-1}| \longleftarrow c_{n-1} \longrightarrow |h_n|$$

This series of connections is called a *connection-chain* [24], whereas the intermediary hosts are called *stepping-stones* [38].

Definition 4.3.1 *A connection-chain C is a list of TCP connections created by recursively logging into a series of hosts;* $C = \langle c_0, \ldots, c_i, \ldots, c_{n-1} \rangle$.

Definition 4.3.2 *Stepping-stones H are intermediary hosts that are used in establishing a connection-chain;*
$H = \langle h_1, \ldots, h_i, \ldots, h_{n-1} \rangle$.

A related notion to consider is the *relative position* of the hosts and connections within a connection-chain. For this purpose, the following terms are defined.

Definition 4.3.3 *In reference to a given host h_j, an* upstream host h_i *is a host that is closer to the* origin *host (h_0); i.e. $0 \leq i < j$. Conversely, a* downstream host h_k *is a host that is closer to the* target *host (h_n); i.e. $j < k \leq n$. The terms* upstream connection *and* downstream connection *are defined similarly.*

[2] In practice, the number of hosts is limited by the maximum delay that a user is willing to tolerate [46].

Yet, another issue to consider is the notion of *directionality*. Although a connection is *bidirectional*, it has a direction in the sense of the *client/server* paradigm [47]. To distinguish each direction, the following definition is provided.

Definition 4.3.4 *Each connection is made of a* forward flow *and a* backward flow. *The* forward flow *refers to the sequence of TCP packets sent by the* client-side *(or simply the* client*), while the* backward flow *refers to the sequence of TCP packets sent by the* server-side *(or simply the* server*).*

At last, it is worth highlighting the following idea. In a connection-chain, each flowing packet has a *header* part and a possible *content* (or *payload*) part. The header part is unique for every single connection in the chain. It can be described by the following 5-tuple:

$$\langle src.ip, src.port, dest.ip, dest.port, protocol \rangle$$

The abbreviations *src* and *dest* stand for *source* and *destination* respectively. They refer to the notion of directionality mentioned earlier.

Unlike the header part, the content part of a packet is relayed across the connections of a connection-chain. Therefore, it should remain the same during its journey through the chain. It, however, can be transformed in its passage depending on the used applications. A common transformation is encryption.

4.3.2 Dynamics of Terminal Applications

Terminal applications have distinctive *functionality* and *traffic pattern*, which set them apart from other applications. Functionality refers to the way that the client-side and the server-side interact during an established session. On the other hand, traffic pattern refers to the characteristics of generated packets as observed at the network-level, such as packets' sizes and inter-arrival times.

In terms of functionality, a terminal application exhibits a distinctive *send/echo* activity between its client-side and server-side. When a user establishes a terminal session and starts typing on her keyboard, the following types of packets are exchanged:

1. *Send packet*: The client *sends* every character as it is being typed by the user.

2. *Echo packet*: The server *echoes* back the sent character in order for the client to display it.

This routine is repeated for every typed character until the user hits the key "*return*" which causes executing the typed command. After the command's execution is finished, the server sends the command's output to the client.

In the case of a connection-chain, the above basic *send/echo* model is stretched over the entire chain. Effectively, the connections are equivalent to a single *logical* TCP connection where a client is located at one end and a server is located at the other end. The individual connections are glued

by in-between stepping-stones. A *send* packet traverses every forward flow, while the corresponding *echo* packet traverses the backward flows.

To stitch the connection-chain, each stepping-stone runs two processes: a server and a client. The server accepts connections from an upstream stepping-stone (or the chain's origin), whereas the client connects to a downstream stepping-stone (or the chain's target). *Send* and *echo* packets are passed over between the two processes inside a stepping-stone. Effectively, *send* packets are pushed downstream towards the target, while the corresponding *echo* packets are pushed upstream towards the origin.

Moving on to the concept of *traffic pattern*, there are generally two classes of network traffic: *interactive* and *bulk transfer*. Terminal applications generate traffic that belongs to the former class. Essentially, the packets flowing in forward flows are dictated by the user's activity instead of the network dynamics. On the contrary, *bulk transfer* sessions (for instance, `ftp` [48]) are limited by factors like TCP flow control, *maximum transfer unit* (MTU), network congestion, etc.

In general, several metrics are used to characterize interactive traffic [49, 50]. For the purpose of studying connection-chains, the following metrics are considered: *packet size* and *packet timing*. Packet size refers to the size of the TCP payload in bytes, while packet timing refers to the characteristics of a packet's arrival/interarrival times.

If the connection is not encrypted (as in `telnet`), a send (and an echo) packet normally has a size of one byte that corresponds to a character typed

by a user. With encryption (as in `ssh`), however, a send (and an echo) packet carries an encrypted version of the typed character. Hence, a packet's size depends on the encryption algorithm used. There are exception cases where few characters may be combined. For instance, `telnet` has a *line* mode where a client sends lines of text instead of individual characters.

For packets' timing, send packets in a forward flow are generated one by one as the user types on her keyboard. As a result, the packets' arrival (and inter-arrival) times are faithful reflections of her typing dynamics rather than the network dynamics. In particular, the inter-arrival times reflect how fast a user can type. There are empirical and statistical models that describe these times rigorously [49, 50]. Also, there are simulation tools to simulate them [51]. We refer the interested reader to the cited publications.

4.3.3 Model and Assumptions

Fortunately, it is not necessary to address the connection-chain problem in a general manner. In practice, there are several *domain knowledge* constraints, which are generally stated as either explicit or implicit assumptions. This section addresses these assumptions.

In principle, connection-chains are solely used for *interactive* attacks. In other words, an attacker uses a connection-chain as a bidirectional channel to interact with a victim machine. This entails two main consequences. First, the TCP protocol is the transport protocol used in connection-chains. Secondly, the underlying traffic is interactive.

It is conceivable for an attacker to use another transport protocol or an unusual covert channel. This, however, requires a great deal of activities such as installing special modules on the stepping-stones. Potentially, such interactions are very loud and easier to detect and trace.

The above discussion leads to the following points:

- TCP [45] is the transport protocol employed in establishing connection-chains.
- A connection-chain can be modeled as a single logical TCP connection that connects a *human* agent at one side to a *victim* machine on the other side.
- Stepping-stones act as relay machines. They however may modify the relayed traffic. Possible modifications include encryption, embedding fictitious packets and adding random or intentional delay. Nonetheless, incoming traffic is related to outgoing one. This sets stepping-stones apart from other types of attack relays; namely *zombies* and *reflectors*.

4.4 Taxonomy of Approaches

In the literature, many approaches have been proposed to detect and/or trace connection-chains. Those approaches integrate numerous interesting ideas and techniques. Before we start exploring them in details, we present a taxonomy that encompasses them. The taxonomy should render a "big

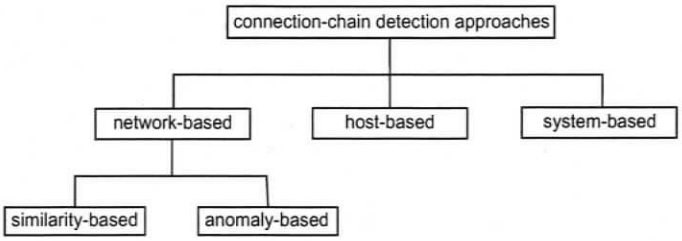

Figure 4.2: Taxonomy of proposed approaches for detecting and tracing connection-chains.

picture" of the proposed approaches.

Generally, connection-chain detection approaches align with the taxonomy shown in Figure 4.2. The top level of classification is based on the familiar *deployment* criteria. In particular, approaches are classified based on the *location* where analysis takes place. Accordingly, approaches that operate on packets at the network level are classified under *network-based* (section 4.5). Also, approaches that function inside hosts are classified as *host-based* approaches (section 4.6). Finally, those approaches, which employ both host-based and network-based components, are classified as *system-based* approaches (section 4.7).

Noticeably, network-based approaches have received most of the research effort. Therefore, we further classify them into either *similarity-based* (section 4.5.1) or *anomaly-based* (section 4.5.2) approaches. In this case, the classification is based on the *nature* of analysis employed in each class.

4.5 Network-Based Approaches

Network-based approaches operate at the network level by examining packets for signs of connection-chains. They are further divided into two main categories; namely *similarity-based* and *anomaly-based*. The difference lies at the scale of analysis. While, similarity-based approaches operate on a set of connections, anomaly-based approaches operate on only a single connection. Each category will now be explained in further details.

4.5.1 Similarity-Based

If two connections c_i and c_j belong to the same connection-chain C, then they are more likely to share some common features. This is especially true for interactive traffic. Therefore, one could devise a *similarity measure* to compare the connections, and flag similar ones as candidates for being part of the same connection-chain. Such measure is typically a function of some *invariant* features that are relayed by the stepping-stones.

Two classes of similarity measures have been proposed in the literature: *content-based* measures and *timing-based* measures. A content-based measure computes similarity by analyzing the packets' contents (payload), whereas the timing-based measure computes similarity by analyzing the packets' timing characteristics.

If the connections are not encrypted, then a content-based measure works well. Basically, a character appearing in a given connection is assured to

appear later in time in either a downstream or upstream connection of the same connection-chain. A naive measure is to simply perform a brute-force text matching between packets' contents [38]. Another simple measure is to trace *unique* strings, for instance a *login* greeting message [38]. Yet, another simple measure is to compute *frequencies* of characters traveling through connections [24].

In addition to introducing the connection-chain problem, Staniford-Chen and Heberlein are first to propose a content-based similarity measure, which they referred to as *thumbprint* [24]. In essence, a thumbprint is a real vector that is computed based on frequencies of characters traversing a connection during a specified time period . It serves as a condensed signature that can be used to differentiate (or relate) two connections.

Regrettably, content-based similarity measures are limited, because they only work if the packets' contents are not encrypted nor modified as they flow through the connection-chain. A more general approach is to correlate connections based on the packets' timings instead of their contents. Collectively, such approaches employ *timing-based* similarity measures. In fact, the majority of connection-chain research belongs to this particular class.

Zhang et al. proposed a simple yet effective timing-based measure [38]. The measure exploits the distinctive ON/OFF patterns observed in interactive traffic. Specifically, observing an interactive connection reveals a pattern of alternating ON and OFF periods. An ON period is when the user is typing on his/her keyboard, while an OFF period is when he/she is idle. The au-

thors devised a similarity measure that computes coincident transition from OFF periods to ON periods among a set of connections. Using such measure, connections with similar transitions are correlated.

Deviation is another timing-based measure proposed by [52]. The measure relies on the following idea. As packets flow through a connection, the total size of transferred bytes tends to increase monotonically in time. Therefore, if two connections belong to the same connection-chain, then their total size of transferred bytes should grow at a similar rate. Obviously, this measure only works if the packet sizes are not altered at the stepping-stones. The authors formally developed this concept and used it to correlate connections.

Wang et al. proposed a timing-based measure that correlates connections based on the inter-packet delay (IPD) in forward flows [53]. In interactive traffic, IPD is a reflection of the typing dynamics of a user. Hence, the authors propose that they are unique and preserved through a connection-chain. They developed a similarity measure to compute and compare connections' IPDs.

Blum et al. proposed a timing-based measure that correlates connections by *counting* packets observed in a time interval [54]. They also showed how many packets are needed to declare whether two connections belong to a connection-chain or not.

He and Tong adapted a signal processing approach to detect connection chains [55]. In this approach, a connection is modeled as a point process, where the points represent the stream of packets in the given connection.

Two connections are part of a connection chain, if their corresponding processes can be shown to exhibit a *casual* mapping (bijection). On the other hand, two connections are not part of a connection chain, if their corresponding processes are shown to be independent. To search for casual mappings between connections, the authors proposed two algorithms. One of the algorithms is timing-based, one that employs a delay constraint to search for possible mapping between incoming and outgoing packets. The other one uses a memory constraint to perform the same task.

Wang and Reeves proposed an active timing-based measure [56]. The idea is to embed a specially designed *watermark* into the flow of packets. If such watermark reappears later in another connection, then the two connections are part of the same connection-chain. The proposed watermark is essentially a modification of inter-packet timing between some selected packets. Peng et al. studied the secrecy of such watermarks and whether they can be detected [57]. They found out that embedded watermarks can be successfully recovered and duplicated if they are not designed carefully. Also, the existence of watermarks can always be quickly detected.

At last, timing-based measures avoid using packets' contents, hence they can be used even if packets are encrypted. They, however, may fail if packets' timing characteristics (for instance arrival times) are disturbed either deliberately or due to the dynamics of the network and hosts.

4.5.2 Anomaly-Based

In similarity-based approaches, the theme is comparing connections using some similarity measure. In contrast, anomaly-based approaches conform to a more *local* approach, where each connection is analyzed in isolation of other connections. Specifically, a connection's forward and backward flows are analyzed. The idea is that *direct* terminal sessions behave differently from *indirect* ones like those comprising connection-chains. In other words, connection-chain manifests a deviation from the *normal* direct terminal session.

This novel approach was first proposed by Yung in [58]. He suggested measuring the following two time estimates:

- *Send-Ack time*: The time taken by a *send* packet to travel to the next host and gets acknowledged. In some respects, this time represents the "hop-to-hop" roundtrip time. Basically, it is an estimate of the *normal* roundtrip time exhibited by a direct terminal session.

- *Send-Echo time*: The roundtrip time for a *send* packet to reach the server side and gets echoed back. In some respects, this time represents the "end-to-end" roundtrip time.

In a direct terminal session, the *Send-Ack* and *Send-Echo* times are expected to be similar. In an indirect terminal session (connection-chain) however, the *Send-Echo* time is expected to be larger than the *Send-Ack* time. In fact, the *Send-Echo* time becomes larger as the connection-chain becomes

longer. There is, however, a catch. Matching a *send* packet with the corresponding *echo* packet can be tricky especially when encryption is used. Yung developed these concepts formally and provided a heuristic matching algorithm.

Yang et al. proposed another anomaly-based algorithm [59, 60]. Their approach suggests analyzing connections in *real-time* as a connection-chain is being established. They proposed several heuristic algorithms to match a *send* packet with its corresponding *echo* packet, in order to measure the *Send-Echo* time in real-time. When a new connection is appended to the connection-chain, the *Send-Echo* time *jumps* to a higher value. A plot of the *Send-Echo* time reveals a step-like function, where every step corresponds to a newly added connection. As such, a step-like behavior is indicative of a connection-chain.

4.6 Host-Based Approaches

In a typical network, a host usually has several inbound and outbound connections that correspond to listening servers and talking clients respectively. If the host is exploited as a stepping-stone, then there must be a correlation between some inbound and outbound connections. In other words, a packet arriving at an inbound connection is assured to reappear in an outbound connection. In network-based approaches, such correlation is ascertained using different similarity measures (see section 4.5.1). In host-based approaches,

however, the story is different.

Typically, all operating systems, by default, do not have a function or a data structure that tells whether an outbound connection has been created by an inbound connection. As a result, one has to actually explore the operating system to find out if such link exists. In the literature, several techniques have been proposed to address this shortage.

One class of techniques employ a process *searching* algorithm based on the following concept. If an outbound connection c_o is created by an inbound connection c_i, then the processes p_i attached to c_i and p_o attached to c_o may be linked. Depending on the operating system, the processes tree can be searched to discover if such link does exist.

Kang et al. proposed a simple search algorithm for a UNIX operating system [61]. Using the notations stated above, the algorithm operates as follows. If c_i and c_o are part of a connection-chain, then p_o (in many cases) is created (`fork`-ed) either directly or indirectly by p_i. Given the fact that in Unix, each process maintains a pointer to its parent process. Then, a simple way to link p_o to p_i is to start at p_o and *recursively* visit its parent process until p_i is found.

The above simple search algorithm fails if the link between p_o and p_i is more involved. For instance, this can be a result of using a pipe or other interprocess communication means. Carrier and Shields proposed a more comprehensive search algorithm to resolve those cases [62]. For each p_o, they proposed walking up the processes tree, exploring a process's parent and all

its siblings. They implemented the algorithm for three Unix-like operating systems; Linux, OpenBSD and Solaris.

Buchholz and Shields proposed a different approach, which does not require searching processes [63]. The approach calls for modifying an operating system to support linking an outbound connection to an inbound one. For each process, a new data structure `origin` is stored in its process table. For processes created by a remote connection, `origin` holds the typical 5-tuple information associated with that connection. For locally created processes, `origin` is undefined. When a process `forks` another one, `origin` is as usual inherited. The authors also proposed other supporting system calls and data structures.

At last, it is worth mentioning that host-based approaches are useful to detect stepping-stones with high accuracy. However, they are not useful by themselves to trace connection-chains. To do so, such approaches have to be employed within a system in order to recursively reveal the whole chain. Also, host-based approaches suffer from an obvious drawback. Specifically, they rely on the *integrity* of the stepping-stones. Such trust can not be established because stepping-stones are compromised hosts by definition.

4.7 System-Based Approaches

In the literature, several system-based approaches have been proposed. In general, they employ an arrangement of collaborating components that to-

gether cooperate to detect and trace connection-chains. The components are both host-based and network-based.

One of the first proposed systems in this class is called *Distributed Intrusion Detection System* (DIDS) [64]. It consists of distributed host/LAN *monitors* and a centralized analysis module called the *director*. In essence, monitors collect auditing data and send them to the director for analysis.

DIDS has an interesting feature that enables tracing a user as he/she moves across a monitored network. The idea is to assign every user a unique network identification (NID) when he/she first logs in the monitored network. An NID is different from the typical user identification (UID). A user may have several UIDs for different hosts and resources, but only a unique NID. Accordingly, a user's activities (including logins) are associated with a single NID by the director. Based on its records, the director can then track a user's movement across the network.

In some respects, DIDS employs a *centralized* paradigm (the director) to trace connection-chains. In contrast, Jung et al. proposed a fully distributed system called *Caller Identification System* (CIS) [65]. The system requires installing two modules at each host: an extended version of *tcp-wrapper* (*ETCPW*) [66] and a CIS server (*CISS*). These modules interact locally and remotely using a distributed protocol to verify the origin of an inbound connection before allowing it in. Connections with inconsistent route information are denied.

Under CIS, a connection-chain $< h_0, \ldots, h_i, \ldots, h_n >$ is recursively traced

as follows. When a new connection arrives at a host h_i, the local $ETCPW$ intercepts it and contacts the local $CISS$ to verify its origin. The local $CISS$, in turn, contacts the $CISS$ at host h_{i-1} requesting route information about the new connection. The remote $CISS$ replies with a list of the previous hosts in the chain; i.e. $< h_0, \ldots h_{i-2} >$. The local $CISS$ then contacts every host in the returned list to verify its integrity. If the integrity test is passed, it saves the list for future requests by the next hosts in the connection-chain; i.e., $< h_{i+1}, \ldots, h_n >$. It, finally, replies back to the local $ETCPW$ to allow the connection in.

Session Token Protocol (STOP) is another fully distributed system that allows to recursively trace connection-chains [62]. In essence, STOP is an enhanced version of the standard *Identification Protocol* (IDENT) [67]. It adds forensics and tracing functionality to IDENT in two essential ways. First, a STOP server is capable of saving user-level and application-level data associated with an outbound connection upon the request of a downstream host. The data is kept locally for future forensic investigation. Secondly, a request can be recursively propagated back to upstream hosts allowing tracing connection-chains. This latter feature is somehow similar to the recursive operation in CIS.

Wang et al. proposed another distributed system that calls for installing special modules at routers as well as modified servers at hosts [68]. This system employs an active approach. Basically, the servers (like `telnetd`) are modified to inject a *watermark* into backward flows upon request. Modules at

the routers detect a watermark and respond appropriately. A watermark is a specially designed string of characters that depends on the modified server.

At last, system-based approaches are projected as a comprehensive solution. They are also meant to attain the best of two worlds; network-based and host-based. They are however expected to be more costly in terms of installation, operation and maintenance.

4.8 Assessment of Approaches

We have reviewed several connection-chain approaches and classify them into several categories. In this section, we evaluate those approaches against a set of criteria. Particularly, the following criteria are considered:

- Domain: The different approaches are applicable to certain domains but not to others. For concrete treatment, we assess their applicability against the following three reference network models suggested in [41] (see section 4.2.3): *closed model, academic model* and *Internet model.* As indicated earlier, the challenges progressively increase as one moves from one model to the next one. Hence, ideally, a technique should work in the most general model; i.e., the Internet model.

- Scalability: An important aspect of a given approach is its ability and flexibility to meet growth demands. We refer to such aspect as scalability, and evaluate the different approaches as *good*, *average* or *poor*. In

principle, network-based approaches have better scalability than host-based ones. Additionally, a central entity is an indication of poor scalability.

- Tracing Ability: Some approaches are designed to only detect connection-chains. Others, however, are capable of tracing as well. In theory, a detection module is needed within a tracing system to initiate a tracing task. We use this criterion to indicate whether a given approach is designed for tracing. We use *yes* or *no* for that.

- Detection Accuracy: Detecting connection-chains is an important aspect of any approach. We use this criterion to rate the detection accuracy[3] for each approach. High accuracy is obviously a desired feature. The following scale is used in this rating: *high*, *average* and *low*.

- Evasion: This criterion refers to the ability of an attacker (who is possibly aware of the system) to escape detection and/or tracing. Ideally, an approach should be immune to evasion attempts. Depending on how hard it is to evade a given approach, the following scale is used: *hard*, *average* and *easy*.

- Cost: Different approaches incur varying costs. Such costs are the result of different factors such as installation, operation, and maintenance. This criterion is an estimate of the aggregate costs incurred by

[3]Detection accuracy is analogous to the *true-positive* rate in intrusion detection systems (IDS).

a given approach. The following scale is used: *high*, *average* and *low*.

Using the above criteria, the assessment is summarized in table 4.1. We use descriptive labels such as high and low to establish a relative comparison between the different approaches. It is important to note that the comparison is largely subjective due to the lack of any quantitative comparison studies in the literature.

For the application domain, we noticed that none of the proposed approaches applies to the Internet model. This is because all of them require installing special modules in the hosts and/or the network infrastructure. Such access is not granted in the Internet model. We also noticed that network-based approaches are more general than other approaches, because they apply to the academic model for which controlling hosts is not required.

For scalability, network-based approaches obviously have an advantage over other approaches. Host-based approaches are expected to have a poor scalability, because modifications are required for every added host. Yet, system-based approaches are expected to have an average scalability, since they employ both host-based and network-based components.

For tracing ability, we mentioned earlier that anomaly-based and host-based approaches can only detect a connection-chain. This is because they employ local analysis. On the other hand, tracing a connection-chain requires a global analysis to reveal the whole chain. Such ability exists in similarity-based and system-based approaches.

Criteria	Network-based (similarity-based)	Network-based (anomaly-based)	Host-based	System-based
Domain	academic	academic	closed	closed
Scalability	good	good	poor	average
Tracing Ability	yes	no	no	yes
Detection Accuracy	average	low	high	high
Evasion	average	average	easy	hard
Cost	low	low	average	high

Table 4.1: Assessment of the different connection-chain approaches.

For detection, host-based modules are expected to produce more accurate results. Network-based modules are inherently less accurate, because they have to deal with issues such as encryption and delay. In fact, anomaly-based ones are expected to be even less accurate. This is because they rely on time estimates (for instance *Send-Ack*) that are hard to measure accurately.

The difficulty of evasion varies among the different approaches. System-based approaches are harder to evade, since they employ different distributed components. Therefore, if one component fails, there is a better chance that another component would resist the evasion. On the other hand, host-based approaches are easiest to evade, because the hosts are already compromised. Finally, network-based approaches are expected to be in between.

In terms of cost, system-based approaches are expected to incur the highest cost, because of the different distributed components involved. Host-based approaches come next, as the cost is expected to be per host. Network-based approaches are expected to come last, since the cost is associated with the network infrastructure.

4.9 Conclusion and Open Directions

In this review, we surveyed several approaches for detecting and tracing connection-chains. We also classified them according to a proposed taxonomy. The review focused on the *technical* issues. Specifically, we showed how an approach works and highlighted some shortcomings as well. We also

assessed the different approaches against a set of criteria. The review also included relevant background materials and various discussions.

We conclude this review by highlighting some possible directions in connection-chains research. Taking this review as a starting point, the following are some possibilities of where to go from here.

- Simulation Environment: Studying connection-chains would benefit from a powerful and flexible simulation environment. Existing simulation packages (for instance ns-2) need nontrivial modifications to simulate connection-chains. Along this direction, Xin et al. have recently described a promising testbed to simulate and evaluate connection-chains [69].

- Quantitative Comparative Study: A valuable study could be conducting a quantitative comparative study of the proposed approaches. Such study requires a framework where different approaches can be compared according to a set of quantitative criteria. In particular, similarity-based approaches are a good candidate for such study.

- Evasion: It is a possibility for a careful attacker to evade detection and/or tracing. Technically, these evasions are targeting the detection and/or tracing process itself. A valuable study would be to enumerate various scenarios under which proposed techniques can be evaded

- New Approaches: This area of research is relatively new. The review may inspire creating new approaches or combining existing ones. The

solution space is quite large, and many ideas are still undiscovered.

- The Internet Model: We mentioned earlier that none of the proposed approaches applies to the Internet model. The ultimate challenge is to propose a solution for tracing connection-chains in this model.

- Network Forensics: Tracing connection-chains is important for network forensics applications. In particular, it has the potential to help uncovering compromised stepping-stones, the attack's path and even the attacker's origin. Therefore, tracing connection-chains could be projected as an infrastructure for a network forensics system.

In the next chapter, we propose a new host-based approach to detect connection chains. The main drawback of the host-based approaches proposed so far in the literature is that they are operating system specific. Specifically, they need to be re-designed and re-implemented differently for different operating system. Also, it is not obvious if they can be applied to proprietary operating systems such as MS Windows.

Another issues, which was not addressed in the literature, is analyzing connection chains that are distributed over several ip addresses. Analyzing these types of connection chains is more challenging. In chapter 6, we further highlight this problem, and propose a profiling-based framework to analyze them.

Chapter 5

Detecting Connection Chains

In this chapter, we propose a host-based algorithm to detect connection chains. We adopt a black-box approach by passively monitoring inbound and outbound packets at a host, and analyzing the observed packets using association rule mining. We first explain the proposed algorithm in details. We then assess the algorithm using public network traces, and demonstrate both its efficiency and detection capabilities.

In general, the main disadvantage of the host-based approaches proposed so far in the literature is that they are operating system specific [63, 62, 61]. Specifically, they are expected to be re-designed and re-implemented differently for different operating system. Also, it is not obvious if they can be applied to proprietary operating systems such as MS Windows.

The proposed technique avoids being operating system specific, by employing a black-box approach. In essence, inbound and outbound packets

are *passively* monitored to detect if there is a connection-chain. The technique is inspired by concepts from association-rule mining in the data mining literature. It also has the following features:

- Portable: The approach is independent of any operating system.
- Real-time: The algorithm is efficient for real-time processing.
- Preserve Privacy: For a packet, the algorithm only uses its arrival time and its header to operate. It neither stores nor uses its payload.
- Robust: The algorithm resists encoding (encryption/compression), because neither packets' payloads nor their lengths are used in the analysis.
- Relative Interpretation: Instead of a crisp yes/no answer, a *confidence* measure is attached to possible connection-chains. This particular feature enables the setting of a user-defined threshold ($minconf$) in order to reduce false positives.

5.1 Attack Model: Another View

Consider the host shown as a rectangle in Figure 5.1. Suppose that this host is exploited as a stepping-stone, and the connections a and b are part of the corresponding connection-chain. Let a_{in}, a_{out}, b_{in} and b_{out} be the corresponding inbound and outbound flows. Since a and b are bidirectional connections,

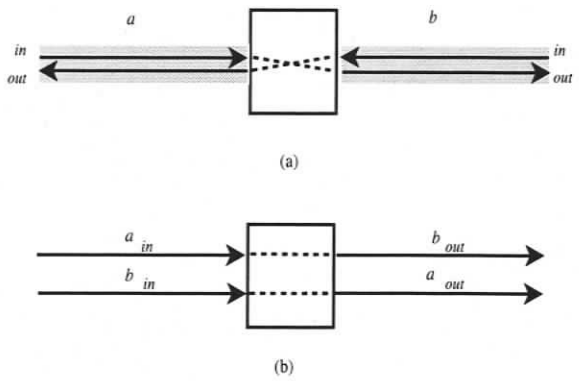

Figure 5.1: Two views of a connection-chain at a host: (a) A connection-view. (b) A flow view.

packets in one connection should re-appear "later" in the other connection. Specifically, an inbound packet coming on a_{in} is expected to reappear as an outbound packet on b_{out}. Similarly, an inbound packet coming on b_{in} is expected to reappear as an outbound packet on a_{out}. The packets' flow inside the host is shown as dotted line. Figure 5.1.(a) depicts a *connection view* of the connection-chain. Figure 5.1.(b) depicts a *flow view*, where inbound and outbound flows are sorted out.

An important question is how soon a packet in one connection will re-appear in another connection if the two connections are part of a connection-chain. Actually, it is not possible to exactly determine this delay ahead of time. However, we know that for a connection-chain to be useful, this delay has to be bounded [46]. In other words, we expect the delay to be random, but not to exceed some constant value. Throughout this thesis, we refer to this constant as Δ.

In reference to the flow view shown in Figure 5.1.(b), our algorithm tries to identify outbound packets that might have been triggered by inbound ones. This is considered the case when the difference between the timestamps of an outbound packet and inbound one is $\leq \Delta$

5.2 Approach Overview

5.2.1 Mining For Association Rules

The proposed approach is based on concepts from the problem of association rules mining. Therefore, we first present a formal statement of this problem before explaining our approach in greater details. The discussion is necessarily brief, however the interested reader may refer to the mentioned citations for more details.

In the field of data mining, *association analysis* is a methodology used to discover interesting relationships in large data sets [28]. The term *association rules* is used to denote the discovered relationships, while the process itself is called *mining for association rules.* Agrawal et al. were first to introduce the methodology, and demonstrate its usefulness in analyzing a database of sales transactions (*market basket transactions*) [70].

The following formulation is adopted from [28]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of n *items.* Let $T = \{t_1, t_2, \ldots, t_N\}$ be a set of N transactions, where each *transaction* t_i contains a subset of items from I, i.e. $t_i \subseteq I$. An *itemset* is defined as a set of items. A *k-itemset* is an itemset that contains k items.

For instance, $\{bread, milk, eggs\}$ is a 3-itemset.

A transaction t_i is said to contain an itemset X, if $X \subseteq t_i$. An important property of an itemset is its *support count*, which refers to the number of transactions that contain a particular itemset. Mathematically, the support count $\sigma(X)$ of an itemset X is given by the following formula:

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$$

where $|.|$ denotes the number of elements in a set.

An *association rule* is an implication of the form $X \rightarrow Y$, where X and Y are disjoint itemsets, i.e. $X \bigcap Y = \phi$. The quality of an association rule is measured by its *support* and *confidence*, which are defined as follows:

$$support \equiv s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{N} \tag{5.1}$$

$$confidence \equiv c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \tag{5.2}$$

Intuitively, the support implies that X and Y occur together in $s\%$ of the total transactions. On the other hand, the confidence implies that, of all the transactions containing X, $c\%$ also contain Y.

It should be noted that there are alternative indices to measure the quality of association rules besides the mentioned *support-confidence framework*. For an account of various indices, the interested reader may refer to [28]. Also,

it is important to note that the implication in an association rule does not necessarily mean *causality*. It simply indicates a co-occurrence relationship between the items in the antecedent and consequent of the rule.

The problem of association rules mining can be formally stated as follows:

Definition 5.2.1 Association Rule Mining: *Given a set of transactions T, find all the rules having support $\geq$ minsup and confidence $\geq$ minconf, where minsup and minconf are user-defined support and confidence thresholds respectively.*

5.2.2 Mining For Connection-Chains

In our approach, we closely follow the classic formulation of association rules mining that was presented in the previous section. In this instance, the items of interest correspond to a set of connections, while the connection chains correspond to the desired association rules. Additionally, a confidence measure is used to denote the strength of a particular connection chain.

Formally, let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of active *connections* at a given host. Also, let $T = \langle t_1, t_2, \ldots \rangle$ be a *sequence* of transactions, where a *transaction* is restricted to one of the following two classes of transaction types:

- *A 1-tuple Transaction*: A transaction of the form $[c_i]$, where $c_i \in C$.
- *A 2-tuple Transaction*: A transaction of the form $[c_i, c_j]$, where $c_i \neq c_j$ and $c_i, c_j \in C$. In this type of transaction, the order is not significant;

i.e. $[c_i, c_j] = [c_j, c_i]$.

Unlike the original formulation, the transactions here are generated dynamically as packets flow in the connections. As such, we refer to the collection of transactions as a sequence instead of a set. A 1-tuple transaction $[c_i]$ is generated whenever an *inbound* packet is received on the corresponding connection. On the other hand, a 2-tuple transaction $[c_i, c_j]$ is generated whenever an *inbound* packet in one connection is followed by an *outbound* packet in the other connection within a Δ amount of time. We will provide more details about the generation of these transactions when we discuss the proposed algorithm in the next section.

In theory, for a set of n connections, there are n 1-tuple transaction types, and $\binom{n}{2}$ 2-tuple transaction types, where the symbol $\binom{\ }{\ }$ denotes the *combination* (*choose*) operator. For example, let the set of connections be $\{a, b, c\}$. Then, there are 3 1-tuple transaction types: $[a], [b]$ and $[c]$. Also, there are 3 2-tuple transaction types: i.e. $[a, b], [a, c]$ and $[b, c]$.

In our formulation, we make a distinction between a transaction type t_i and how many times t_i actually occurred. We use the *support count* $\sigma(t_i)$ to refer to how many times a transaction of type t_i has occurred. For example, let the set of connections be $\{a, b, c\}$. Then, we might have the following support counts: $\sigma([a]) = 10, \sigma([b, c]) = 2, \ldots$ etc.

As mentioned earlier in the introduction, a connection chain appears at a host as a pair of connections through which traffic flows back and forth. As a result, in our framework we view a *connection-chain* as an association rule

of the form $\{c_i, c_j\}$, where $c_i \neq c_j$ and $c_i, c_j \in C$. Note that a set notation is used to represent a connection chain instead of an implication ($\rightarrow$). This is to emphasize the fact that a connection chain does not imply any direction. For instance, given a connection chain $\{c_i, c_j\}$, packets are investigated in both directions; i.e. $c_i \rightarrow c_j$ and $c_j \rightarrow c_i$.

The confidence of a connection-chain $\{c_i, c_j\}$ is defined as follows:

$$confidence(\{c_i, c_j\}) = \frac{\sigma([c_i, c_j])}{\sigma([c_i]) + \sigma([c_j])} \tag{5.3}$$

where $c_i \neq c_j$ and $c_i, c_j \in C$.

Intuitively, the numerator represents those times when packets in both connections occur within Δ amount of time, while the denominator represents the times they occur solely. Typically, a true connection chain is expected to have a high confidence close to 1.

To understand the logic behind this formula, note that the classic confidence measure (equation 5.2) is an *asymmetric* one; i.e. the confidences of $A \rightarrow B$ and $B \rightarrow A$ may neither be the same nor related. In our formulation, however, we need a confidence measure that combines both. In particular, consider the two connections a and b and the possible connection chain $\{a, b\}$. We like to account for packets flowing in both directions: $a \rightarrow b$ and $b \rightarrow a$. Specifically, we like to investigate the following two association rules:

$$a_{in} \rightarrow b_{out}$$

$$b_{in} \rightarrow a_{out}$$

where *in* and *out* stand for inbound and outbound packets respectively.

According to the classic definition of confidence (equation 5.2), the confidences of these two association rules are given by:

$$confidence(a_{in} \rightarrow b_{out}) = \frac{\sigma(a_{in} \cup b_{out})}{\sigma(a_{in})}$$

$$confidence(b_{in} \rightarrow a_{out}) = \frac{\sigma(b_{in} \cup a_{out})}{\sigma(b_{in})}$$

Recall that a 1-tuple transaction $[c_i]$ is generated whenever an *inbound* packet is received on the corresponding connection, and a 2-tuple transaction $[c_i, c_j]$ is generated whenever an *inbound* packet in one connection is followed by an *outbound* packet in the other connection within a Δ amount of time. As such, the denominators in these two confidences correspond to the support count of 1-tuple transactions $[a]$ and $[b]$ respectively. Also, the *sum* of the numerators is equal to the support count of 2-tuple transactions $[a, b]$. Therefore, the above confidences can be restated as follows:

$$confidence(a_{in} \rightarrow b_{out}) = \frac{\sigma([a, b])^1}{\sigma([a])}$$

$$confidence(b_{in} \rightarrow a_{out}) = \frac{\sigma([a, b])^2}{\sigma([b])}$$

where $\sigma([a, b])^1$ and $\sigma([a, b])^2$ are the support count of 2-tuple transactions $[a, b]$ contributed by each direction.

In order to assess the connection chain $\{a, b\}$, we have to combine these two individual confidences by taking into account the symmetrical nature of a connection chain. The formulation adopted in this thesis is similar to the well-known *Jaccard similarity index* [28]. For reference, the Jaccard similarity index for two sets A and B is given by

$$J(A, B) = \frac{|A \bigcap B|}{|A \bigcup B|}$$

We find out that the proposed formula actually works well, as we show later when we discuss the evaluations of the algorithm in section 6.4.

5.3 Algorithm and Implementation

5.3.1 Data Structures

In the proposed algorithm, we employ the following two main data structures.

- **A Connection-Chain Graph (*ccGraph*)**: An undirected weighted graph $G(N, E, W)$, where
  - a node $n \in N$ represents an active connection c_i. For 1-tuple transactions of type $[c_i]$, the support count $\sigma([c_i])$ is stored here.
  - an edge $e \in E$ exists between two nodes v (representing a connection c_i) and u (representing a connection c_j), if there are packets suspected to be flowing between the two connections. An edge's

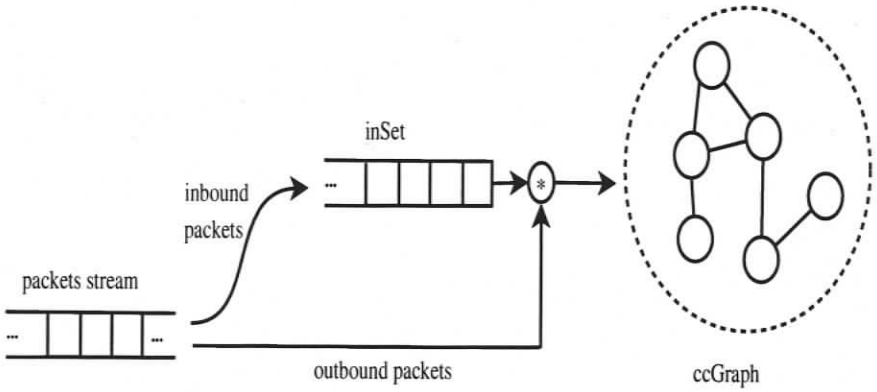

Figure 5.2: Overall flow of the algorithm and used data structures.

weight $w \in W$ corresponds to the support count $\sigma([c_i, c_j])$ for the 2-tuple transactions of type $[c_i, c_j]$.

- **Inbound Packets Set (*inSet*)**: A set of *current* inbound packets. When a new inbound packet is received, it is added to this set. For a particular connection, *inSet* contains only the most recent inbound packet on that connection. In some respects, this set acts as a time-sliding window of the current inbound packets. The size of the window is approximately Δ time unit.

Figure 5.2 depicts an overall flow of the algorithm. It also shows how these data structures fit in the whole picture. In essence, the input to the algorithm is a stream of packets seen at a host's network interface. Inbound packets are buffered into *inSet* for later comparisons. Outbound packets are not buffered. Instead, they are compared with the already buffered inbound packets to determine if there is a correlation. In addition, *ccGraph* maintains

an up-to-date status of active connections and possible correlations between them. In particular, *ccGraph* holds the support counts for the two types of transactions that was discussed earlier.

5.3.2 Algorithm

Algorithm: PROCESSPACKET(p)

```
if d(p) = in then
    increment σ([c(p)]) in ccGraph ;
    add p to inSet ;
else if d(p) = out then
    forall q ∈ inSet do
        if t(p) − t(q) ≤ Δ then
            if c(p) ≠ c(q) then
                increment w(c(p), c(q)) in ccGraph ;
            end
        else
            remove q from inSet ;
        end
    end
end
```

Figure 5.3: The overall algorithm as a pseudocode.

In Figure 5.3, we summarize the whole algorithm as a pseudocode. For each packet $p \in P$, the following operators are defined:

- $t(p)$: the time-stamp of p.
- $c(p)$: the connection to which p belongs.
- $d(p)$: the direction of p; either inbound (in) or outbound (out).

The received packets are processed on a first-come-first-served basis according to their timestamps. Each packet generates a transaction that depends on its direction. An inbound packet p generates a 1-tuple transaction of the type $[c(p)]$. The support count for this type of transaction $\sigma([c(p)])$ is stored into the corresponding node in *ccGraph*. Additionally, the packet itself is buffered into *inSet* as mentioned earlier. Intuitively, an inbound packet either results in incrementing the count of that connection, or creating a new node (connection) if it does not exist. Additionally, it is buffered in *inSet* for later comparison with outbound packets.

On the other hand, an outbound packet p generates a 2-tuple transaction of type $[c(p), c(q)]$, if $q \in inSet$, $c(p) \neq c(q)$, and $t(p) - t(q) \leq \Delta$. The support count for this type of transaction $\sigma([c(p), c(q)])$ is stored as a weight $w(c(p), c(q))$ of the edge between $c(p)$ and $c(q)$ in *ccGraph*. Intuitively, we keep counts of every pair of outbound and inbound packets, if the outbound packet comes after the inbound one by at most Δ.

At any time, connection-chains are edges in *ccGraph* that have a *confidence* exceeding some user-defined threshold (*minconf*). The confidence is computed according to equation 5.3.

5.3.3 Example

In this section, a simple example is presented to demonstrate how the algorithm works in practice. We show how a stream of packets is processed, and how the data structures are updated.

Table 5.1: A simple example to demonstrate the proposed algorithm.

received packet	generated transaction type	*inSet*	*ccGraph*					
			$\sigma([a])$	$\sigma([b])$	$\sigma([c])$	$\sigma([a,b])$	$\sigma([a,c])$	$\sigma([b,c])$
$(a_{in},1)$	$[a]$	$\langle(a_{in},1)\rangle$	1	-	-	-	-	-
$(c_{out},2)$	$[a,c]$	$\langle(a_{in},1)\rangle$	1	-	0	-	1	-
$(b_{out},3)$	$[a,b]$	$\langle(a_{in},1)\rangle$	1	0	0	1	1	-
$(c_{in},4)$	$[c]$	$\langle(a_{in},1),(c_{in},4)\rangle$	1	0	1	1	1	-
$(b_{in},5)$	$[b]$	$\langle(a_{in},1),(c_{in},4),(b_{in},5)\rangle$	1	1	1	1	1	-
$(b_{in},6)$	$[b]$	$\langle(a_{in},1),(c_{in},4),(b_{in},6)\rangle$	1	2	1	1	1	-
$(a_{out},7)$	$[a,b]$	$\langle(b_{in},6)\rangle$	1	2	1	2	1	-
$(a_{out},8)$	$[a,b]$	$\langle(b_{in},6)\rangle$	1	2	1	3	1	-
$(c_{in},9)$	$[c]$	$\langle(b_{in},6),(c_{in},9)\rangle$	1	2	2	3	1	-
$(c_{out},10)$	-	$\langle(c_{in},9)\rangle$	1	2	2	3	1	-

Let $C = \{a, b, c\}$ be the set of active connections at a given host. As packets flow in these connections, we then have the following:

- 3 possible connection chains; i.e. $\{a, b\}, \{a, c\}$, and $\{b, c\}$.
- 3 possible types of 1-tuple transactions; i.e. $[a], [b]$ and $[c]$.
- 3 possible types of 2-tuple transactions; i.e. $[a, b], [a, c]$ and $[b, c]$.

Consider the list of packets shown in table 5.1. In the first column, we list the packets in the order they are processed. For instance, a packet denoted as $(a_{in}, 1)$ means that the packet is an *inbound* packet, belonging to connection a and received at time 1. After processing a packet, we show generated transactions types (if there is any) in the second column. Additionally, the third and fourth columns are used to show the content of *inSet* and *ccGraph* as the algorithm progresses. In particular, *inSet* column shows the buffered packets, while *ccGraph* column shows the support counts for all possible transaction types.

In this example, assume that Δ is set to 2 time units. Consequently, the following is a highlight of the algorithm's operations for each packet:

- $(a_{in}, 1)$: An inbound packet on a is received. Therefore, an $[a]$ transaction is generated. Because this is the first inbound packet on a, a new node in *ccGraph* is created for the connection a and $\sigma([a])$ is set to 1. The packet is then added to *inSet*.

- $(c_{out}, 2)$: An outbound packet on c is received. Therefore, it is compared with packets in $inSet$. In this case, this packet occurs within a Δ time from the buffered packet $(a_{in}, 1)$. As such, an $[a, c]$ transaction is generated, and an edge in $ccGraph$ is created with $\sigma([a, c])$ set to 1. Note that a node for connection c is also created with $\sigma([c]) = 0$.

- $(b_{out}, 3)$: An outbound packet on b is received. Therefore, it is compared with packets in $inSet$. In this case, this packet also occurs within a Δ time from the buffered packet $(a_{in}, 1)$. As such, an $[a, b]$ transaction is generated, and an edge in $ccGraph$ is created with $\sigma([a, b])$ set to 1. Note that a node for connection b is also created with $\sigma([b]) = 0$.

- $(c_{in}, 4)$: An inbound packet on c is received. Therefore, a $[c]$ transaction is generated, and $\sigma([c])$ is incremented. The packet is then added to $inSet$.

- $(b_{in}, 5)$: An inbound packet on b is received. Therefore, a $[b]$ transaction is generated, and $\sigma([b])$ is incremented. The packet is then added to $inSet$.

- $(b_{in}, 6)$: Another inbound packet on b is received. Therefore, a $[b]$ transaction is generated, and $\sigma([b])$ is incremented. The packet is then added to $inSet$. However, notice that the old packet $(b_{in}, 5)$ is replaced by the new one $(b_{in}, 6)$.

- $(a_{out}, 7)$: An outbound packet on a is received. Therefore, it is com-

pared with packets in $inSet$. In this case, this packet occurs within a Δ time from only one of the buffered packet; i.e. $(b_{in}, 6)$. As such, an $[a, b]$ transaction is generated, and $\sigma([a, b])$ is incremented. Additionally, the other packets in $inSet$ are deleted.

- $(a_{out}, 8)$: Another outbound packet on a is received. Therefore, it is compared with packets in $inSet$. In this case, this packet occurs within a Δ time from the buffered packet $(b_{in}, 6)$. As such, an $[a, b]$ transaction is generated, and $\sigma([a, b])$ is incremented.

- $(c_{in}, 9)$: An inbound packet on c is received. Therefore, a $[c]$ transaction is generated, and $\sigma([c])$ is incremented. The packet is then added to $inSet$.

- $(c_{out}, 10)$: An outbound packet on c is received. Therefore, it is compared with packets in $inSet$. In this case, this packet does not match with any of the buffered packets. In particular, $(b_{in}, 6)$ occurs before $(c_{out}, 10)$ by more than Δ time unit, and $(c_{in}, 9)$ belongs to the same connection. As such, no transaction is generated here. Additionally, the packet $(b_{in}, 6)$ is deleted from $inSet$.

After processing those packets, there are two detected connection chains; namely $\{a, b\}$ and $\{a, c\}$. The confidences of each one is given by:

$$confidence(\{a, b\}) = \frac{\sigma([a, b])}{\sigma([a]) + \sigma([b])} = \frac{3}{1 + 2} = 1$$

and

$$confidence(\{a, c\}) = \frac{\sigma([a, c])}{\sigma([a]) + \sigma([c])} = \frac{1}{1+2} = 0.33$$

5.3.4 Implementation

We implemented the algorithm in Java 1.5 [71]. We also used a Java binding of the *pcap* library, called *Jpcap* [72]. Concerning the data structures, *ccGraph* was implemented using two hash tables; one for the nodes (connections), and one for edges (connection-chains). *inSet* was also implemented using a hash table.

5.4 Evaluation

5.4.1 Data and Settings

We used a public network trace to assess the algorithm in terms of processing time and detection capability. The trace is called *LBNL-FTP-PKT*, and is available from [73]. The trace contains all incoming anonymous FTP connections (i.e. to port 21) to public FTP servers at the Lawrence Berkeley National Laboratory during a ten-day period in Jan 10-19, 2003. It contains 3.2 million packets flowing in 22 thousand connections. The connections are between 320 distinct servers and 5832 distinct clients.

We chose this trace for two reasons. First, it is reasonably large to assess the algorithm. Secondly, it only contains the interactive part of ftp sessions;

i.e. the control stream, on which commands are passed to the FTP server [48]. This type of traffic is similar to interactive traffic generated by terminal applications such as `telnet` [42], `rlogin` [43] and `ssh` [44]. In particular, this is the type of traffic seen in connection-chains.

It is important to note that this trace does not contain any connection chains. Therefore, to assess the detection capabilities, we simulated some connection chains and merged them into the trace as we show later.

To start with, we sliced the trace into 320 subtraces by server ip address. Each subtrace contains the packets exchanged with the corresponding server. Then, we run the algorithm on those subtraces, as if the algorithm was running on the corresponding server.

By analysing inbound and outbound packets of those servers, we estimated the response time of the servers to be between 10-90 msec. We used this value as a guidance to set Δ in our test suite. In particular, Δ is varied as 1, 10, 50, 100, 200, and 500 msec. The selected values are intended to investigate the effect of varying Δ, when Δ is set *below*, *around*, and *above* the estimated true Δ value.

Finally, the tests were run on a laptop with the following specifications: a 1.3Ghz Intel Pentium m-processor, 2 GB RAM, and 80 GB 7200 RPM Hard drive.

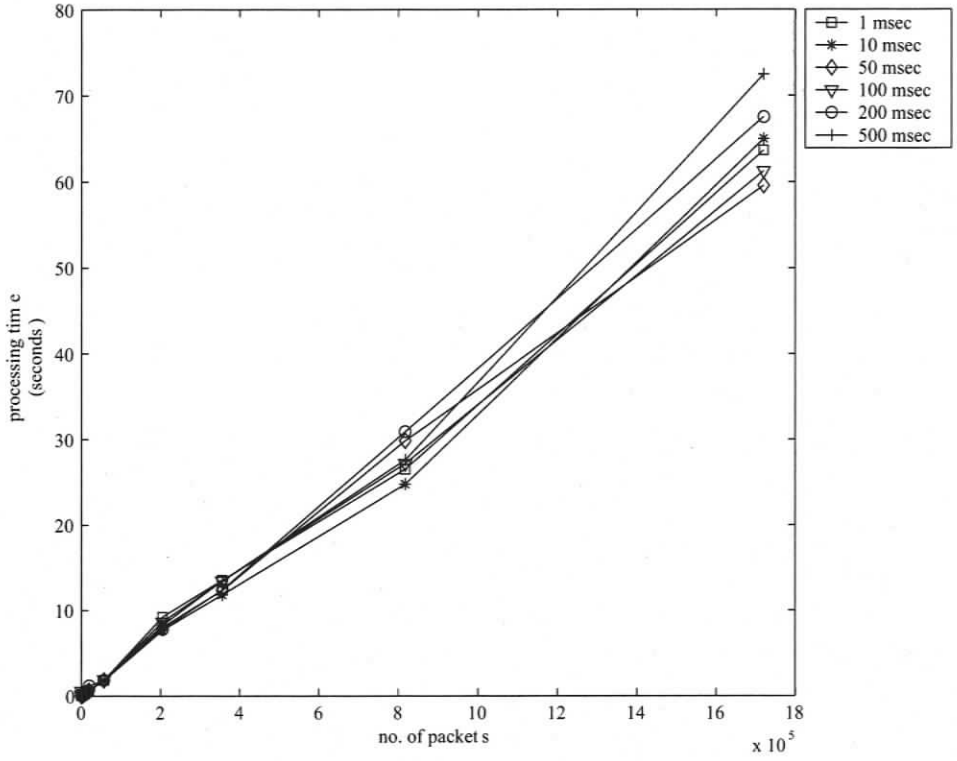

Figure 5.4: The processing time of the 320 subtraces sorted in increasing order of the number of packets. Note that the processing time exhibits a linear trend as subtraces increase in size, and also that varying Δ does not significantly impact on the processing time.

5.4.2 Processing Time

The proposed algorithm is intended for real-time processing of a live stream of packets. Therefore, we are interested in investigating the processing time per packet, and how does this time scales as a subtrace increases in size. We are also interested in studying how the processing time is affected by varying Δ.

The experimentation is performed as follows. For every subtrace (320 subtraces), we run the algorithm with a Δ of 1, 10, 50, 100, 200, and 500 msec; i.e. a total of $6 \times 320 = 1920$ cases. For a particular subtrace S_i, we then observe the processing time T_i. The results are plotted in Figure 5.4.

In Figure 5.4, we notice the following points:

- The processing time exhibits a linear trend as subtraces increase in size.
- The processing time per packet is constant. It basically corresponds to the slope of the lines. Mathematically, it is given by $\frac{T_i}{|S_i|}$ $seconds/packet$, where $|S_i|$ is the number of packets.
- For this trace, the average processing time per packet is about 35 μsec/packet.
- Finally, varying Δ does not seem to have a significant effect on the processing time.

Based on the above observations, we conclude that the algorithm is efficient for real-time operation. This is because the average processing time per packet is both constant and low.

5.4.3 Detection

To assess the detection capabilities of the algorithm, we simulated some connection chains, because the trace does not contain any. We then merged the simulated connection chains into the original subtrace and run the algorithm.

As in the previous section, we also varied Δ here. We then made note of the detected connection chains and their confidences.

The process of simulating connection-chains is depicted in Figure 5.5. In this figure, L, R and R' respectively stand for the local host, a remote host and a *fictitious* remote host. The steps to create a simulated connection chain {R,R'} are as follows:

- For an inbound packet (R,L), create an outbound packet (L,R'). The time-stamp of the new packet is set to original time-stamp **plus** some random time t.
- For an outbound packet (L,R), create an inbound packet (R',L). The time-stamp of the new packet is set to original time-stamp **minus** some random time t.
- Merge those generated packets into the original trace.

Following the above steps, packets would seem to be flowing between the two remote hosts R and R'. In the figure, original packets are shown as solid arrows, while the simulated ones are shown as dotted arrows. Obviously, a table is maintained to keep a consistent mapping between hosts R and R'.

Consequently, the experimentation is performed as follows. At first, we picked the largest subtrace among the 320 subtraces. This particular subtrace was chosen because of its large size, although other subtraces give similar results. The subtrace has 1.7 millions packets flowing in 3391 connections. The traffic is exchanged between the server (131.243.2.12) and 236 unique remote

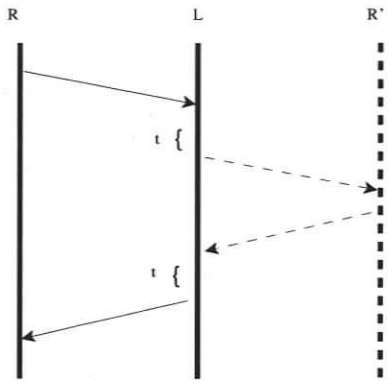

Figure 5.5: The process of simulating a connection chain. L, R and R' respectively stand for the local host, a remote host and a *fictitious* remote host. Original packets are shown as solid arrows, while the simulated ones are shown as dotted arrows.

clients (addresses). Among these 236 unique remote addresses, we randomly picked 88 of them to create simulated connection chains according to the steps explained earlier. For the random time t, we use a uniform random variable between 10-90 msec (an estimate of the server response time). Accordingly, the modified subtrace has $236 + 88 = 324$ remote addresses and $\binom{324}{2} = 52326$ possible connection chains. Only 88 out of the 52326 possible connection chains are *true* connection chains ($\approx 0.2\%$). Those are the ones that we actually simulated.

The modified subtrace is then used as an input to the algorithm. In order to study all connection chains detected by the algorithm regardless of their confidences, we compute confidence statistics for different values of Δ. The following values of Δ were considered: 1, 10, 50, 100, 200, and 500 msec. Note that a Δ of 100 msec is the ideal value in this case, because the server

response time is estimated to be 10-90 msec.

A summary of the the algorithm's output is shown in Table 5.2. For each value of Δ, we list several descriptive statistical quantities to show the confidences distributions about the true and false connection chains involved in the evaluation dataset. For true connection chains, there are 88 possible ones. For false connection chains, there are $\binom{324}{2}$ - 88 = 52238 possible ones. To evaluate the actual detection capability of the algorithm we need to select a $minconf$ value. We set $minconf$ to 0, in order to study all connection chains detected by the algorithm regardless of their confidences; $minconf = 0$ actually corresponds to the worst case scenario. Figure 5.6 and Table 5.3 illustrate the true and false detection rates under such scenario. The true detection rate (TDR) corresponds to the percentage of detected true connection chains in relation to all possible true connection chains, while the false detection rate (FDR) is the percentage of detected false connection chains in relation to all possible false connections.

Considering the number of detected connection chains, we notice the following points. First, the number generally increases as Δ increases. This is actually expected because increasing Δ would increase a connection's buffering time in $inSet$. Consequently, a connection would have greater chance to correlate with other connections. However, notice that the rate of this increase slows down as Δ gets larger than the ideal Δ (100 msec). For instance, increasing Δ from 100 to 500 msec results in less than 1% increase in the number of false conection chains. Secondly, notice that setting Δ to

Table 5.2: A summary of The Algorithm's output showing confidence statistics for different values of Δ under any non negative value for $minconf$.

		Confidence					
		Min	1st Quartile	Median	Mean	3rd Quartile	Max
$\Delta = 1$ ms	True	0.01429	0.01857	0.02389	0.02639	0.0317	0.04348
	False	0.0002823	0.0007423	0.0009671	0.001242	0.00151	0.0122
$\Delta = 10$ ms	True	0.02439	0.03584	0.06797	0.07214	0.08378	0.1923
	False	0.0003401	0.001433	0.002322	0.002967	0.003913	0.02817
$\Delta = 50$ ms	True	0.2581	0.4756	0.4093	0.4929	0.5595	0.8077
	False	0.0003804	0.002959	0.006042	0.009131	0.01292	0.07726
$\Delta = 100$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0003623	0.004518	0.01006	0.01599	0.02237	0.1467
$\Delta = 200$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0003623	0.008181	0.01796	0.03009	0.04302	0.2653
$\Delta = 500$ ms	True	1.0	1.0	1.0	1.0	1.0	1.0
	False	0.0004968	0.01471	0.03562	0.05958	0.08803	0.4173

Table 5.3: True and False Connection Chain Detection Rates under Worst Case Scenario: $minconf = 0$

Δ (ms)	1	10	50	100	200	500
TDR(%)	4.5	31.8	100.0	100.0	100.0	100.0
FDR(%)	1.0	4.2	6.9	7.8	8.2	8.7

a very low value can result in missing true connection chains. For instance, only 4.5% and 32% of true connection chains were detected when Δ is set to 1 and 10 msec respectively. Based on these two points, we conclude that it is safe to set Δ to a higher value than the true one in order to detect all true connection chains, especially that the increase in false connection chains is not significant.

Considering the confidences data in Table 5.2, true connection chains generally have higher confidences. In Figure 5.7, the *ranges* (min-max) of confidences are visualized. For each value of Δ, a band is shown that spans the range of all possible confidences. Inside each band, the grey region indicates the range for false connection chains, while the black region indicates the range for true connection chains.

In Figure 5.7, notice how the confidences of true and false connection chains overlap when Δ is set to very low values (1 and 10 msec). However, once Δ is set around or above the ideal value, true connection chains are clearly separated. In this case by appropriately setting the confidence threshold ($minconf$) in the separation area, we achieve perfect detection rates. For instance, for $\Delta = 100$ msec, by setting $minconf = 0.5$ we obtain TDR = 100% and FDR = 0%. Also, notice that increasing Δ beyond the

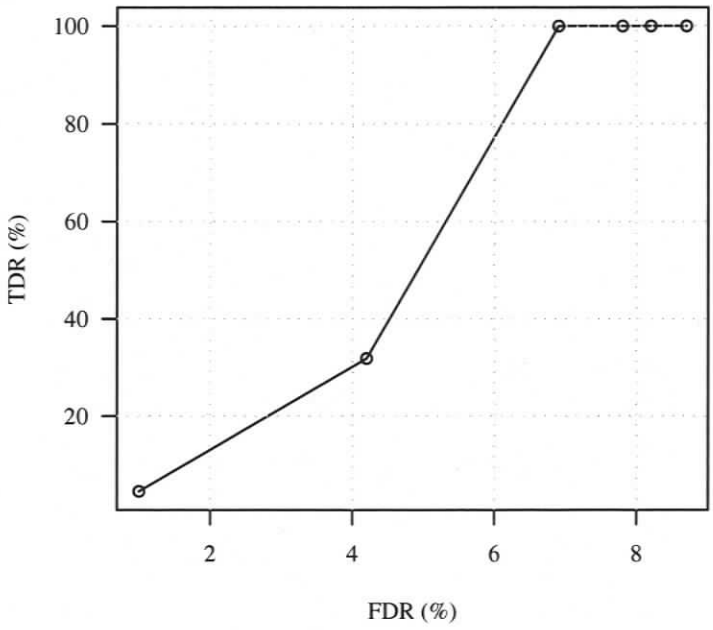

Figure 5.6: ROC curve showing how the TDR and FDR vary when different values are used for Δ, under worst case scenario ($minconf = 0$).

ideal value decreases the *separation* between the confidences of the true and false connection chains. In this case, the maximum separation occurs at the ideal value of Δ (100 msec). However, notice that this separation is reasonably large even when $\Delta = 500$ msec; i.e. 5 times the ideal value. In essence, large separation is desirable because it gives greater flexibility in setting the $minconf$ threshold. Such threshold is used to reduce (or eliminate) false connection chains.

Considering the confidences of false connection chains alone, we notice

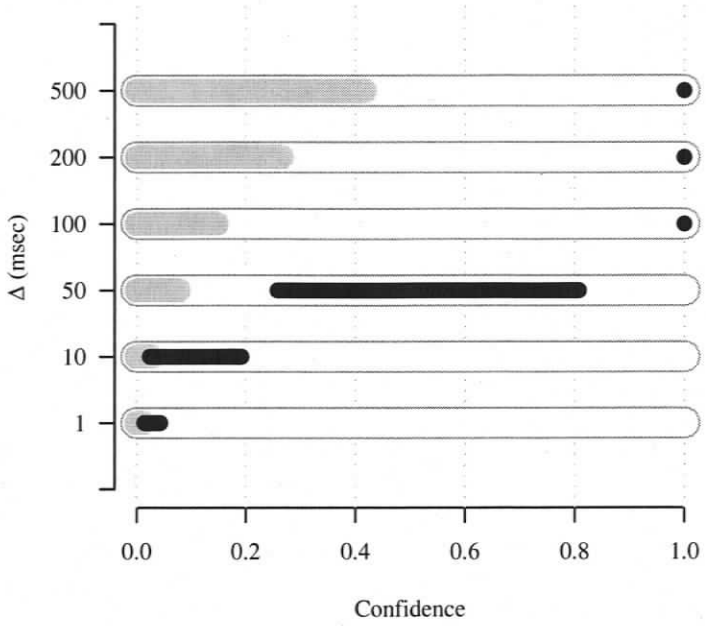

Figure 5.7: The ranges (min-max) of confidences of true and false connection chains for different values of Δ. For each value of Δ, a grey region indicates the range for false connection chains, while a black region indicates the range for true ones.

that they exhibit a similar distribution. In Figure 5.8, the distributions are shown as a set of histograms for each value of Δ. They approximately follow a decaying exponential distribution. In other words, the majority of them have confidences close to zero, while few have larger confidences. In fact, this is a desirable feature because the majority of false connection chains can be eliminated using a low $minconf$ threshold.

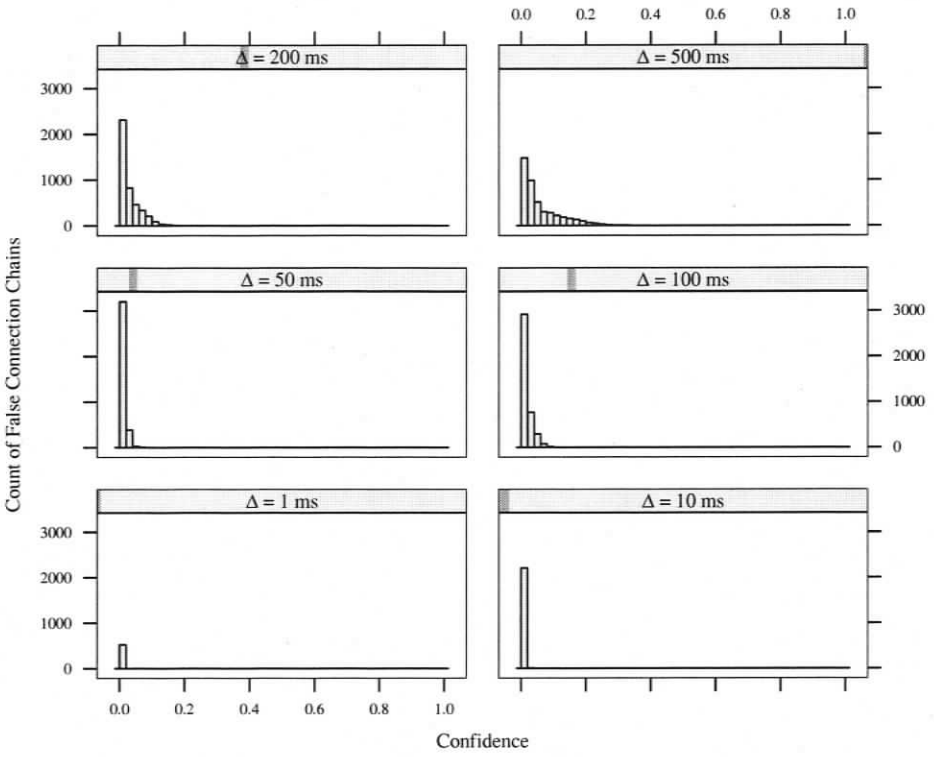

Figure 5.8: The confidences of false connection chains are shown as a set of histograms for different values of Δ. They approximately follow a decaying exponential distribution.

5.5 Evaluation using Real Attack

In this section, we evaluate the proposed framework using a real attack that was posted as a forensic challenge by the Honeynet project [20]. The attack was presented as challenge#18 in a series of 34 challenges called *scan of the month challenges.*

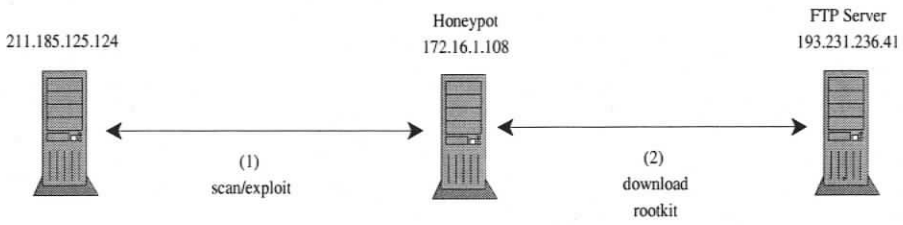

Figure 5.9: A diagram showing the ip addresses involved in the attack, and the attack steps.

5.5.1 Attack Overview

For this attack, Honeynet project provided a file of captured traffic to/from a honeypot. The file is in the standard *pcap* format [10]. In total, there were 993 packets communicated between the honeypot and 7 remote ip addresses. Among the 7 ip addresses, two ip addresses were involved in a successful attack against the honeypot. Specifically, one ip address was used to scan and compromise the honeypot, and one ip address was accessed to download a rootkit.

The steps of the attack and the involved ip addresses are shown in Figure 5.9. Briefly, the attack proceeded as follows:

1. 211.185.125.124 scanned the honeypot for rpc service (port 111), and confirms that *statd* is running. It then executed a successful exploit(rpc.statd) which bound a root shell at port (39168). It finally connected to the honeypot (via port 39168), and obtained a root shell.

2. Via the obtained shell, the attacker connected (ftp) to an ftp server on

Table 5.4: The confidences of the true connection chain for different values of Δ.

$\Delta(ms)$	1	10	50	100	200	500
Confidence	-	0.3335	0.3666	0.406	0.4391	0.7644

193.231.236.41, and downloaded/installed a rootkit.

5.5.2 Results

Since the captured traffic contains 7 ip addresses, there are, technically, $\binom{7}{2}$ = 21 *possible* connection chains. Among these 21 possible connection chains, there is only one *true* connection chain, which is depicted in Figure 5.9.

We run the algorithm on the original trace provided by Honeynet project. As in the previous section, we also vary Δ here, and consider the same values of Δ; namely 1, 10, 50, 100, 200 and 500 msec. Also, $minconf$ is set to 0 here in order to study all connection chains detected by the algorithm regardless of their confidences. Then, for every Δ, we record the detected connection chains and their corresponding confidences.

In this attack, the algorithm successfully revealed the true connection chain with a false detection rate (FDR) of 0%. Except for $\Delta = 1$ msec, the algorithm only detects one connection chain which is the true connection chain depicted in Figure 5.9. The confidences of this connection chain for different values of Δ are shown in Table 5.4. For $\Delta = 1$ msec, the algorithm do not detect any connection chains.

5.6 Conclusion

In this chapter, we proposed a host-based algorithm to detect connection chains by passively monitoring inbound and outbound packets. From a host perspective, a connection chain appears as a pair of connections through which packets pass back and forth. We took advantage of the fact that the time taken by a packet inside the host has to be bounded for a connection chain to work. We refer to this time as Δ. As such, we employed concepts from association rule mining in the data mining literature. In particular, we proposed efficient algorithm and data structures to discover connection chains among a set of connections. Also, a confidence measure is used to attest the likelihood of a connection chain.

We used public network traces to assess the algorithm in terms of processing time and detection capabilities. For processing time, our experimentations suggest that the algorithm is efficient for real-time operation. It has a constant and low average processing time per packet. In terms of detection capabilities, our experimentations suggest that the algorithm is effective in detecting true connection chains. The setting of Δ seems to play an important role. We found that it is always safe to set Δ to a higher value than the true value. This ensures the detection of all true connection chains, while the increase in false connection chains is not significant. Also, our experimentations showed some desirable features of the algorithm. In particular, the majority of false connection chains have confidences close to zero, while few

have larger confidences. This means that the majority of false connection chains can be eliminated using a low confidence threshold. Also, we found that the confidences of true and false connection chains are clearly separated when Δ is set around or above (even 5 times) the true value. This also gives greater flexibility in setting a confidence threshold to reduce (or eliminate) false connection chains.

Chapter 6

Profiling Connection Chains

Investigating connection chains can be complicated when several ip addresses are used in the attack. In this chapter, we highlight this challenging problem. We then propose a solution through hacker profiling. Our solution includes a novel hacker model that integrates information about a hacker's linguistic, operating system and time of activity. It also includes an algorithm to operate on the proposed model.

6.1 Connection Chains as Multistage Attacks

Generally, an attack is either considered a *single-stage* isolated attack or a step in a *multi-stage* attack scenario. Most *intrusion detection systems* operate with the first assumption [74]. On the other hand, investigating attacks as a multi-stage scenario is the central focus of research in the area of

alert correlation [75] and *network forensics* [76]. In some respects, connection chains can be viewed as a multistage attack.

A multi-stage attack scenario can be further studied from two perspectives; namely *temporal* and *spatial*. From a temporal perspective, an attack's steps are assumed to be originating from a single ip address. Additionally, each step of the attack follows the next one in time. In the spatial view, however, an attack's steps are distributed over more than one ip address. Most work in the area of alert correlation is concerned with the temporal perspective.

We illustrate the consequences of such limitation with two scenarios. The first scenario is depicted in Fig 6.1.a. In this example, two independent hackers use the same intermediary host to attack a victim machine. In this case, those approaches would mistakenly aggregate/correlate data from two different hackers and hence two different attacks.

Figure 6.1.b depicts the second scenario. In this example, a hacker uses two different intermediary hosts for his attack. The victim will see two different ip addresses and has no way to tell that they are originating from the same hacker. Consequently, data and alerts from the two ip addresses would pass uncorrelated.

To address this problem, we need a way to profile hackers in order to differentiate one from another. Specifically, we need to look for invariant features that may distinguish one hacker from another, instead of only relying on his ip address. By examining received packets, we believe that it is possible

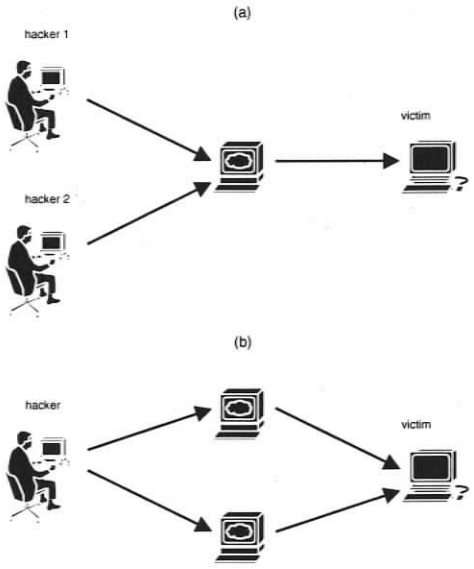

Figure 6.1: Two scenarios which confuse a correlation process. (a) Two independent hackers use same intermediary node to attack a victim. (b) One hacker use two intermediary nodes to attack the same victim.

to extract such features.

One of these features is a hacker's operating system (OS). OS *passive fingerprinting* is a well established technique that can be used for OS identification by examining the headers of certain packets (especially SYN packets) [35].

We propose a solution that enhances the OS passive fingerprinting [35] by using two additional features; namely a hacker's *linguistics* and *activity time*. For linguistics, the content of a packet is examined to look for clues about a hacker's language. For instance, a user/password combination may reveal a name that is known to belong to a certain language.

For time of activity, a hacker may have a preferred time to carry out his actions. By measuring his activities during a day, we may obtain a consistent pattern that further enhances the profiling process. We use arrival time of packets to determine this feature as we will show later.

The main contributions in this chapter are (1) introducing the *spatial* factor of network hacking activities. Such factor affects both alert correlation and network forensics approaches, (2) proposing a simple yet extensible hacker model to address the mentioned spatial factor, and (3) providing a framework/algorithm where the mentioned hacker model is applied.

6.2 Model Formulation

6.2.1 A Hacker Model

Our model for a hacker is defined as a 3-tuple $\mathcal{H}$ over the Cartesian product $\{\mathcal{O} \times \mathcal{L} \times \mathcal{T}\}$ where:

- $\mathcal{O} = \{x : x \text{ is the name of a known operating system}\}$
- $\mathcal{L} = \{x : x \text{ is the name of a known natural language}\}$
- $\mathcal{T} = \{Morning, Evening, Night\}$

$\mathcal{O}$ is basically a set of nominal values corresponding to different operating systems; such as {Windows98, Linux, ...}. Similarly, $\mathcal{L}$ is a set of nominal values corresponding to different natural languages, such as {English, French, ...}. Lastly, $\mathcal{T}$ is a tri-valued variable that indicates when a hacker is most active. Instead of recording detailed time information, we use this variable to classify a hacker as either a morning, evening, or night hacker. This provides an enormous reduction in data and a reasonable granularity for our purpose.

6.2.2 Model Application

Applying our model to a stream of incoming packets is essentially an abstraction process by which we intend to map every ip address into a hacker

instance. Given a list of ip addresses P, this is formally stated as follows.

$$P \rightarrow \mathcal{H} : \{\mathcal{O} \times \mathcal{L} \times \mathcal{T}\}$$

In order to execute this mapping, we need the following three mapping functions:

$$\begin{cases} \psi \equiv header(p) \rightarrow \mathcal{O} \\ \lambda \equiv content(p) \rightarrow \mathcal{L} \\ \phi \equiv arrivalTime(p) \rightarrow \mathcal{T} \end{cases}$$

where, $header(p)$, $content(p)$ and $arrivalTime(p)$ are, respectively, the headers, contents and arrival times of packets originating from an ip address p.

The ψ function applies OS passive fingerprinting techniques in order to infer the OS of an ip address p [35]. Typically, the header portion of an ip packet is used for such task. Formally, this function is a trained classifier that operates by comparing certain header's fields to a database of known OS signatures. For example, the 8-bit *time-to-live* (TTL) field in the ip header is set with different values depending on the operating system. By default, for instance, TTL is set to 128 in Windows and 64 in Linux. Such features can be used to distinguish between different operating systems.

The λ function assigns an ip address p to one or more natural languages. This task is known as language guessing in the field of *text categorization* [77]. A popular technique, that accomplishes this task with almost-perfect accuracy, is due to Cavnar and Trenkle [78]. Briefly, the technique relies on

analysing the n-grams[1] of a document. The frequencies of different n-grams provide distinctive signatures that not only distinguish documents, but also discriminate between languages. There is already an efficient implementation of the algorithm as well as signatures for more than 70 different natural languages [79]. Formally, this function is also a trained classifier.

Finally, the ϕ function uses packets' arrival times to classify a hacker as either a morning, evening, or night hacker. Essentially, it is a 3-category histogram function of the packets' arrival times. In reference to local time, the function divides a day into three 8-hours time periods as shown below:

$$\overbrace{4am \ldots 12pm}^{\text{morning}} \ldots \overbrace{12pm \ldots 8pm}^{\text{evening}} \ldots \overbrace{8pm \ldots 4am}^{\text{night}}$$

Accordingly, it counts how many packets received in each period then yields the matching period with the maximum count. Note that this day division is in reference to the local time zone, rather than hackers' time zone.

6.2.3 Coherence

After performing the above mapping, an ip address falls into one of two categories: *coherent* ip addresses or *incoherent* ones. A coherent ip address is an ip address that assumes a single value for each variable in the hacker model. On the contrary, an incoherent ip address is an ip address which assumes multiple values for one or more of the variables in the hacker model.

[1] An N-gram is an N-character slice of a longer string [78].

To illustrate this concept, assume two ip addresses p_1 and p_2 are mapped as follows:

$$p_1 \rightarrow \{\{Windows98\} \times \{English\} \times \{Morning\}\}$$
$$p_2 \rightarrow \{\{WindowsXP, Linux2.4\} \times \{Arabic\} \times \{Night\}\}$$

In this example, p_1 is a coherent ip address because it assumes single values for each variable. However, p_2 is an incoherent ip address because the OS variable assumed two values; namely $WindowsXP$ and $Linux2.4$.

6.3 Approach

6.3.1 Overview

Using our model and the defined functions, we obtain a mapping for every single ip address into a hacker instance; i.e. $P \rightarrow \mathcal{H} : \{\mathcal{O} \times \mathcal{L} \times \mathcal{T}\}$. With this in hand, we can now address the two problems depicted in Figure 6.1.

For the first scenario (fig 6.1.a), the victim will receive packets from an incoherent ip address. For instance, the packets may reveal multiple values for the OS variable. Hence, the victim will be able to detect such ip addresses.

The second scenario (fig 6.1.b) is more challenging. In this case, we need a way to group ip addresses that share the same attributes. This task can be done using *clustering analysis* [80]. The details of this step is postponed to the next section.

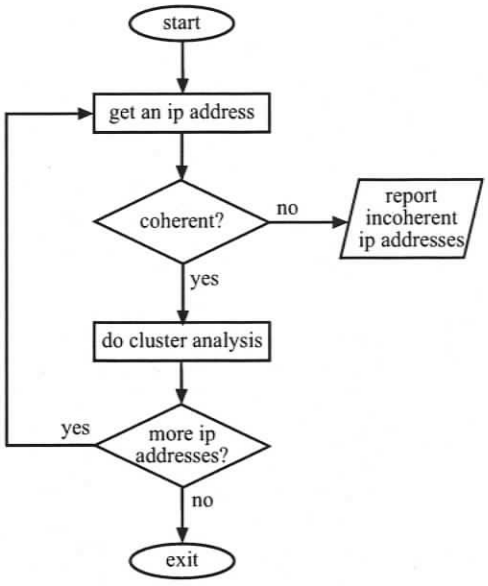

Figure 6.2: Flowchart of overall approach.

A flowchart of the overall approach is shown in Figure 6.2. The algorithm iterates over each ip address. An ip address is first tested for coherence. Incoherent ip addresses are reported as being possible intermediary node shared by multiple different hackers. On the other hand, coherent ip addresses are passed to a clustering analysis algorithm. The algorithm determines groups of ip addresses that are likely used by one hacker.

6.3.2 Clustering Analysis Algorithm

In order to carry the clustering analysis, a measure of *dissimilarity* between observed objects is required. This measure is typically dependent on the type of variables that describe an object. In our case, the objects are coherent

ip addresses which are described by the three variables of our hacker model. Additionally, the variables are *multistate nominal* ones. We now summarize the derivation of the measure following Gordon [80].

For each variable, a disagreement indices can be defined between each pair of states of the variable. Let $\delta_{klm}(\geq 0)$ be the disagreement between the lth and mth states of the kth variable. Accordingly, let $\delta_{klm} = 1$ if $l \neq m$ and $\delta_{kll} = 0$. The contribution to the dissimilarity d_{ij} between the ith and jth object that is made by the kth variable is defined by $d_{ijk} = \delta_{klm}$ if the kth variable is in state l for the ith object and state m for the jth object. The overall measure of dissimilarity between the ith and jth objects is then defined as

$$dis(i,j) \equiv \sum_{k=1}^{K} d_{ijk}$$

where K is the total number of variables.

After defining a dissimilarity measure, we need an algorithm to perform the clustering analysis [29, 80]. For our purposes, we devised an *incremental* algorithm shown in fig 6.3. The algorithm incrementally classify every coherent ip address. Based on the dissimilarity measure, an ip address is either combined into an existing cluster or put in a new cluster. By the end, each cluster will contain one or more coherent ip addresses that share the same attributes; i.e. likely coming from the same hacker.

It should be noted that the number of clusters is dynamically updated as new clusters are formed. This departs from algorithms (*k-means* for instance)

Algorithm: DOCLUSTERING(C_0, P)

Input: a list of initial clusters C_0.
Input: a list of ip addresses P.
Output: a list of resulting clusters C.

```
C ← C_0 ;
foreach p ∈ P do
    clustered ← false ;
    i ← 1 ;
    while (not clustered) and (i ≤ |C|) do
        if dis(p, C(i)) == 0 then
            C(i) ← C(i) + {p} ;
            clustered ← true ;
        end
        i ← i + 1 ;
    end
    if (not clustered) then
        |C| ← |C| + 1 ;
        C(|C|) ← {p};
    end
end
return C ;
```

Figure 6.3: A pseudocode of the clustering algorithm. Note that $|C|$ stands for the size of C, and $C(i)$ stands for the i^{th} element in C.

that require specifying the ultimate number of clusters for effective analysis.

6.3.3 Example

We conclude this section with an example to illustrate our approach. Suppose that a victim PC was attacked by 4 different ip addresses $P = \{p_1, p_2, p_3, p_4\}$.

Also, suppose they are mapped as follows:

$$p_1 \rightarrow \{\{Windows98\} \times \{English\} \times \{Evening\}\}$$
$$p_2 \rightarrow \{\{WindowsXP\} \times \{Arabic, Chinese\} \times \{Night\}\}$$
$$p_3 \rightarrow \{\{FreeBSD\} \times \{French\} \times \{Morning\}\}$$
$$p_4 \rightarrow \{\{Windows98\} \times \{English\} \times \{Evening\}\}$$

The algorithm first identifies p_2 as an incoherent *ip* address because more than one language were detected. This means that p_2 is probably a node used by several hackers similar to the scenario depicted in fig 6.1.a.

The other three ip addresses are coherent. Therefore, they are advanced for the clustering analysis algorithm. At the end of the analysis, we will have 2 clusters; i.e. $\{p_1, p_4\}$ and $\{p_3\}$. The $\{p_1, p_4\}$ cluster probably represents a situation similar to the scenario at fig 6.1.b. While, the $\{p_3\}$ cluster contains only one ip address, which means that it is probably a single hacker using a single ip address.

6.4 Simulations

We conducted various simulations to assess the proposed model and the clustering algorithm. For this purpose, we employed empirical probability distributions of operating systems and languages that reflect typical usage on the Internet. In particular, we used the freely available distributions shown in Figure 6.4. For the time variable $\mathcal{T}$, we used a discrete uniform

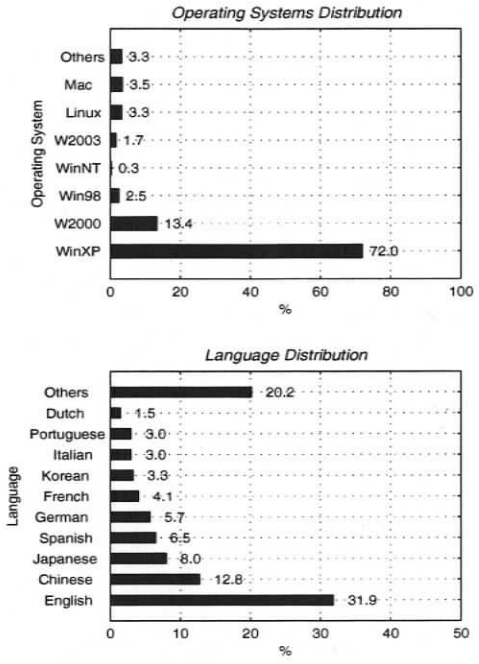

Figure 6.4: Empirical probability distributions of operating systems and languages on the Internet. (OS source: www.w3schools.com, Language source: www.internetworldstats.com.)

distribution. We then generated many random samples of ip addresses that have characteristics following these distributions. All simulations were run using `Matlab` version 6.5 on a 1.3GHz laptop with 512MB.

6.4.1 Time Complexity

To start with, we evaluated the time complexity of the clustering algorithm. Specifically, we evaluated the running time of the algorithm in *batch mode* and in *incremental mode*. In batch mode, the algorithm is passed a set of ip addresses with no prior clusters. The algorithm then builds clusters starting from scratch. In contrast, in incremental mode, the algorithm is passed one ip address at a time along with a set of initial clusters.

In practice, the batch mode is intended for offline analysis, while the incremental mode is intended for realtime analysis. The distinction between the two modes depends on the parameters that are passed to the algorithm. In batch mode, the call is DOCLUSTERING($\{\}, P$), where the parameters are an empty set of initial clusters and a set of ip addresses P. In incremental mode, however, the call is DOCLUSTERING($C_0, \{p\}$), where the parameters are an initial set of clusters C_0 and a set of only one ip address $\{p\}$. Typically, the algorithm is first run in batch mode to form an initial set of clusters C_0, then it is run in incremental mode to process new ip addresses as they arrive.

At a first glance, the algorithm may suggest a quadratic running time in terms of the total number of ip addresses because of the double loop.

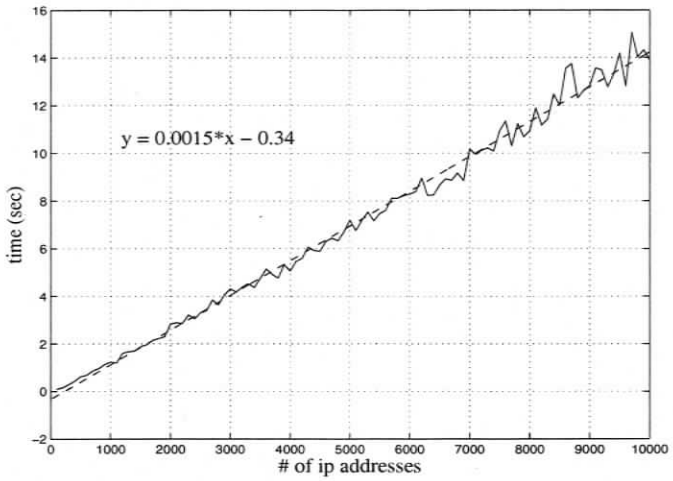

Figure 6.5: Execution time of the clustering algorithm as a function of the number of ip addresses.

However, our simulations consistently showed a linear running time in batch mode, and a small running time in incremental mode.

For batch mode, we generated 100 samples with sizes that increase from 1 up to 10000 at an increment of 100. Each sample was then passed to the clustering algorithm. The time taken by the algorithm was then recorded. To ensure the validity of the results, 5 reruns of the same procedure were performed. In Figure 6.5, we show a plot of a typical run. As shown, the execution time increases linearly as the number of ip addresses increases. For reference, the linear fitting (dashed line) is also shown in the figure. It should be noted that 10000 is relatively very large, because this number is supposed to be the number of unique "ip addresses" attacking a victim at one time.

In incremental mode, the outer loop is invoked only once because there is

only one ip address to process. Therefore, the running time is basically the time taken by the inner loop to scan the initial clusters, i.e. $O(C_0)$. In our experiments, we found this time to be very small. For instance, the running time for clusters with up to 10000 ip addresses never exceeded 0.01 seconds.

6.4.2 Effectiveness

To a great extent, the effectiveness of our approach relies on the underlying distributions of the ip addresses. In this section, we present simulations related to this aspect.

At first, we studied incoherent ip addresses and their expected rates relative to the total population. We generated 100 samples with sizes that increase from 1 up to 10000 at an increment of 100. Each sample was then analyzed for incoherent ip addresses. An ip address is regarded incoherent when either the operating system or the language is set to "others". We then used the following metric:

$$incoherence\ rate = \frac{i}{N}$$

where i is the number of incoherent ip addresses in the population, and N is the total number of ip addresses in the population.

A plot of the incoherence rate is shown in Figure 6.6. It is interesting to note that the rate is almost constant as the size of the population increases. In this simulation, the average rate is about 0.23. In other words, about

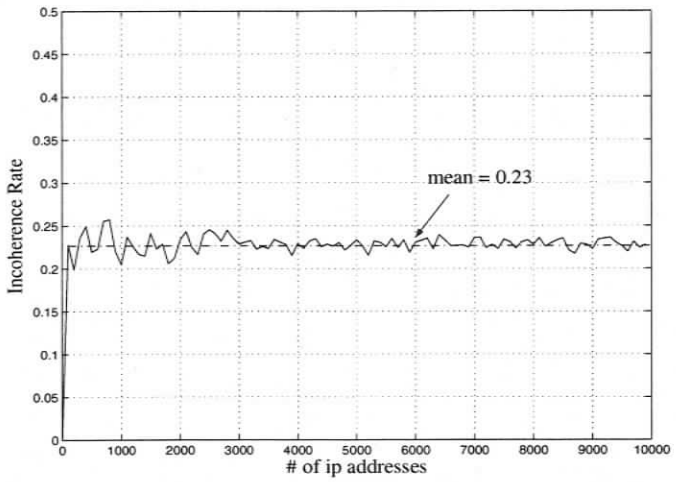

Figure 6.6: The rate of incoherent ip addresses relative to the total population.

23% of the population is expected to be incoherent and dropped from the clustering analysis.

We next considered the clustering of coherent ip addresses. It should be obvious that the number of possible clusters depends on the *resolution* of each variable in our model. We use the term "resolution" here to denote the number of possible states that a variable may assume. To show this graphically, we used the following metric:

$$clusters\ rate = \frac{nc}{N}$$

where nc is the number of clusters and N is the total number of ip addresses.

We generated 50 samples with sizes that increase from 1 up to 500 at an increment of 10. A plot of the clusters rate is shown in Figure 6.7. As

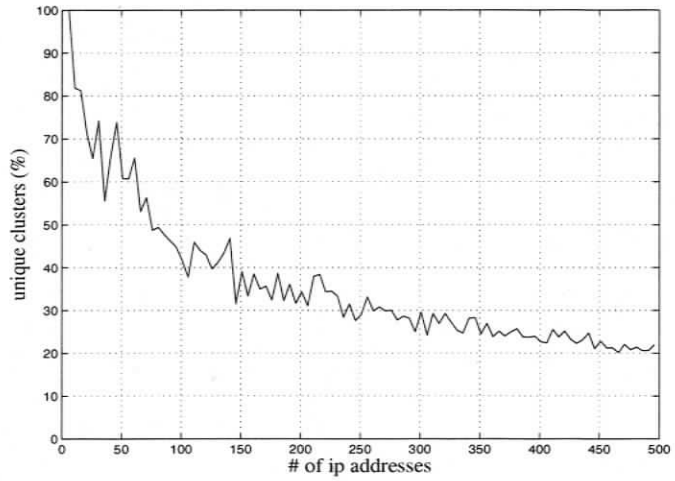

Figure 6.7: The ratio of the number of unique clusters as the population of ip addresses increases.

shown, the ratio decreases rapidly (almost exponentially) as the number of ip addresses increases. Therefore, the algorithm will always deal with limited number of clusters. Also, this confirms that the algorithm has a low overhead as was shown in the previous section.

6.5 Evaluation using Real Attack

In this section, we evaluate the proposed framework using a real attack that was posted as a forensic challenge by the Honeynet project [20]. The attack was presented as challenge#19 in a series of 34 challenges called *scan of the month challenges*. We carefully picked this particular attack, because it is the only one that fits the attack scenarios addressed by our framework. In particular, it resembles the scenario depicted in Figure 6.1 (b).

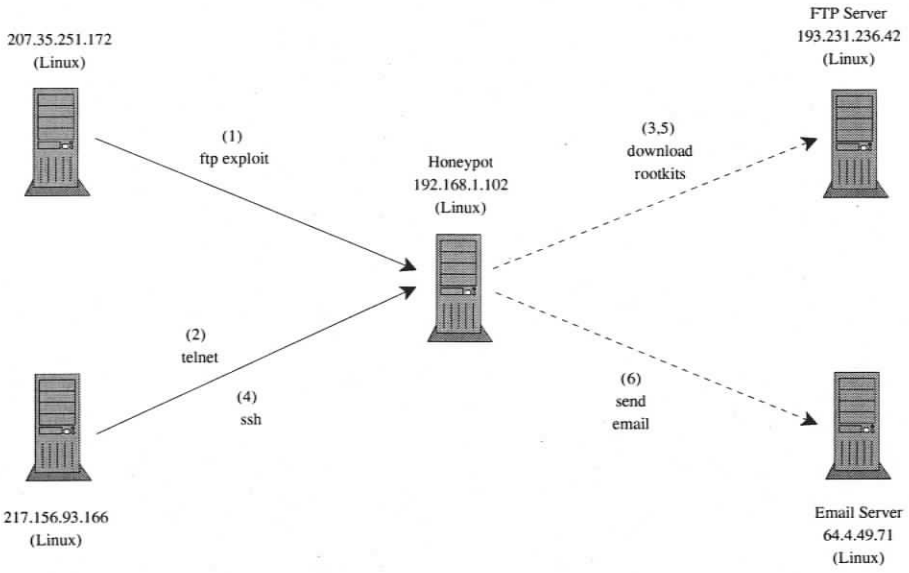

Figure 6.8: A diagram showing the ip addresses involved in the attack, and the attack steps.

6.5.1 Attack Overview

For this attack, Honeynet project provided 2 files in *pcap* format [10]. One file (newdat3.log) contains captured network traffic to/from a honeypot. The second file (slog2.log) contains generated syslog traffic. In total, there were 24440 packets communicated between the honeypot and 16 remote ip addresses. Among the 16 ip addresses, four ip addresses were involved in the attack. Specifically, two ip addresses were actually used to compromise the honeypot, one ip address was accessed to download rootkits, and one was used to send an email.

The steps of the attack and the involved ip addresses are shown in Figure

6.8. Briefly, the attack proceeded as follows:

1. 207.35.251.172 connected (ftp) to the honeypot, run a successful exploit, and obtained a root shell. He then deleted the password of user "nobody", and added a new user (dns) with root privilege.

2. 217.156.93.166 connected (telnet) to the honeypot using user "nobody", and obtained a shell.

3. Via the obtained telenet shell, the attacker connected (ftp) to an ftp server on 193.231.236.42, and downloaded/installed a rootkit. Among other things, the rootkit started a modified ssh server on port 24 (backdoor).

4. 217.156.93.166 connected (ssh) to the honeypot, and obtained a shell.

5. Via the obtained ssh shell, the attacker again connected (ftp) to the ftp server on 193.231.236.42, and downloaded/installed more rootkits.

6. An email is sent through the email server at 64.4.49.71.

To see how this attack fits the scenario in Figure 6.1 (b), notice that the attack was concurrently carried from two ip addresses; namely 207.35.251.172 and 217.156.93.166. Apparently, the attacker controlled the computers at these ip addresses. The other two ip addresses (193.231.236.42 and 64.4.49.71), however, were mainly used to download rootkits and send an email. As such, their links are indicated as doted lines.

6.5.2 Analysis

In order to apply our framework, each ip address has to be mapped into a hacker instance; i.e. a 3-tuple {OS, Language, Activity Time }. The mapping was done using an automated script as follows.

- For OS mapping, we used an open source tool called p0f [35]. P0f is a versatile *passive* OS fingerprinting tool, which can identify the operating system using a number of techniques.

- For the activity time, we built a small script that classifies each ip address as either morning, evening or night. The script simply counts the packets exchanged with each ip address and reports when they occur most.

- For Language mapping, we extracted readable text using a Unix tool called *strings*. We, then, examined the text manually to determine the language. We had to resort to manual examination, because we could not find a suitable tool. In future, we intend to build such tool.

We show the resultant mapping in Table 6.1. The ip addresses involved in the attack occupy the 4 bottom rows of the table, and are indicated with asterisks (*). For each ip address, we also show the packet count, and explaining remarks. These remarks are mainly to explain why the language was (was not) detected. Finally, we indicate missing values (OS, Language) as "-".

Table 6.1: Mapping of the ip addresses in challenge#19 from Honeynet project. (*) indicates the ip addresses involved in the attack.

ip address	packets count	OS	language	Activity Time	Remarks
66.51.200.115	1	Windows	-	evening	No payload.
208.179.195.130	2	Linux	-	night	No payload.
24.248.173.56	2	Linux	-	morning	No payload.
138.86.152.104	6	-	-	night	No payload.
128.175.106.247	10	-	-	night	Port scan only. No payload.
207.50.37.225	10	-	-	morning	No payload.
63.168.30.92	10	-		morning	No payload.
210.114.220.46	11	Linux	-	night	No payload.
206.75.218.84	11	FreeBSD	-	evening	No payload.
64.58.76.226	12	-	-	evening	No payload.
24.17.45.29	28	-	-	morning	No payload.
207.245.82.221	88	-	-	evening	Port scan only. No payload.
64.4.49.71*	29	Linux	-	evening	Payload contains commands/responses.
193.231.236.42*	543	Linux	Romania	evening	Romanian name (password) detected.
217.156.93.166*	2455	Linux	Romania	evening	Romanian name (password) detected.
207.35.251.172*	21222	Linux	-	evening	Payload contains commands/responses.

For some ip addresses, it was not possible to detect the OS fingerprints. The reason is related to the techniques used by the tool p0f. In particular, p0f detects OS by examining only TCP packets (SYN, SYN+ACK and RST/RST+ACK). Other packets, for instance UDP and ICMP, are ignored. In our case, the undetected ip addresses actually exchanged non-TCP traffic. To the best of our knowledge, there is no tool that can OS fingerprint using non-TCP traffic. This could be due to technical difficulty or due to the prevalence of TCP traffic. Creating such a tool, however, is definitely a welcome contribution.

For language detection, Romanian language was associated with two of the ip addresses involved in the attack (217.156.93.166 and 193.231.236.42). In particular, a Romanian name (gunoierul) was used as a password in an ftp session. For other ip addresses, it was not possible to detect the language because of the lack of payload, or because of having a payload which contains only commands and their responses.

6.5.3 Discussion

In the proposed framework, clustering analysis is used to aggregate mapped ip addresses into groups that share the same attributes. In reference to Table 6.1, we can identify two groups that share 2 or more attributes:

- Group 1: {208.179.195.130, 210.114.220.46}.
- Group 2: {64.4.49.71, 193.231.236.42, 217.156.93.166, 207.35.251.172}.

For group 1, the two ip addresses have the same OS and activity time; i.e Linux and night respectively. However, this grouping can not be confirmed, because only few packets are communicated (2 and 11 packets respectively). Also, the communicated packets carry no payload! Therefore, the relationship in this group is inconclusive and requires further investigation.

On the other hand, the ip addresses in group 2 are, indeed, related. They are all involved in the attack as explained earlier. We also can be certain this time, because many packets were communicated by ip addresses in this group.

On the downside, we notice that OS and Language fingerprinting generate many missing values. In this case, the language was detected in two out of 16 ip addresses (12.5%), and OS detected in 9 out 16 ip addresses (56.25%). Technically, this is due to the used tools, rather than the proposed framework. However, we believe that adding more attributes into the proposed hacker model can further enhance the profiling process.

6.6 Conclusions

Investigating multi-stage attack scenarios can be complicated when *several* ip addresses are used in the attack. In this chapter, we introduced this *spatial* factor of network hacking activities. Such factor affects both alert correlation and network forensics approaches. We also proposed a simple yet extensible hacker model to address the mentioned spatial factor. The model

integrates information about a hacker's linguistic, operating system and time of activity. The mentioned model is then applied within a framework. We finally conducted several simulations and an evaluation with real data to assess our approach.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Network forensics is a young member of the bigger family of digital forensics discipline. In particular, it refers to digital forensics in networked environments. It represents an important extension to the model of network security where emphasis is traditionally put on prevention and to a lesser extent on detection. It focuses on the *collection*, and *analysis* of network packets and events caused by an intruder for investigative purposes.

A key challenge in network forensics is to ensure that the network itself is forensically-ready. In particular, a network must have an infrastructure to collect and analyze data in real-time. The infrastructure should ensure that the needed data exists for a full investigation. After all, no investigation can be performed if data does not exist. Because of networks' distributed nature,

designing such infrastructure is a challenging task. One, for instance, has to decide what/where data should be collected?

In chapter 3, we proposed an agent-based network forensics system, which is intended to add real-time network forensics capabilities into a controlled network. The proposed architecture employs different agents to support the distributed nature of computer networks and to separate the collection, logging and analysis processes.

Another challenge in network forensics arises because of attackers ability to move around in the network. In particular, it is common for a computer attack to originate from an attacker's computer, propagate through other computers, then attack a victim computer. This ability to move around in the network results in creating a chain of connections; commonly known as *connection chains*. They are widely used by attackers to stay anonymous and/or to confuse the forensic process.

In chapter 4, we presented an extensive survey of proposed approaches for detecting and tracing connection-chains. We also classified them according to a proposed taxonomy. We also assessed the different approaches against a set of criteria. The review also included relevant background materials, various discussions and suggestions for future directions.

In chapter 5, we proposed a novel framework to detect connection chains. We adopted a black-box approach by passively monitoring inbound and outbound packets at a host, and analyzing the observed packets using association rule mining. We assessed the proposed framework using public network

traces, and demonstrated both its efficiency and detection capabilities.

Finally, in chapter 6 we proposed a profiling-based framework to investigate connection chains that are distributed over several ip addresses. The framework utilizes a simple yet extensible hacker model that integrates information about a hacker's linguistic, operating system and time of activity. Several simulations were conducted to assess the framework.

7.2 Future Work

In the future, we believe that the following areas would greatly improve this work:

- The proposed network forensic system adds a very effective forensic infrastructure to a network. We, however, believe that it can be improved in a number of ways:
  - For the current design, communication between the different agents is not standardized. We did not impose a particular communication protocol between these agents. Therefore, more work is needed to devise (or adapt) a communication protocol between all agents. We believe that peer-to-peer (P2P) protocols are promising candidates.
  - In this work, we proposed two analysis agents to detect and profile connection chains. We believe that more analysis agents are

needed in order to further automate the forensic process.

  - Further scalability studies are needed to assess the proposed architecture. In particular, it is valuable to evaluate this architecture on a large-scale network.

- One of the major obstacle in evaluating algorithms for detecting connection chains is the lack of a benchmark, and/or appropriate datasets. We believe that developing such a benchmark will prove valuable to advance this line of research.

- A valuable extension is to identify attack scenarios, which can be used to evade the proposed frameworks. Accordingly, countermeasures of such scenarios are developed to further improve these frameworks.

Bibliography

[1] Robert Richardson. 2003 csi/fbi computer crime and security survey. Technical report, CSI, 2003.

[2] Computer Crime & Intellectual Property Section. United states department of justice. `http://www.usdoj.gov/criminal/cybercrime/cccases.html`. (Last visited: May 26, 2007).

[3] Marcus Ranum. Network forensics: Network traffic monitoring. Technical report, Network Flight Recorder, Inc., 1997.

[4] Gary Palmer. A road map for digital forensic research. In *Digital Forensic Research Workshop*, Utica, New York, 2001.

[5] U.S. Department of Justice: National Institute of Justice. *Electronic Crime Scene Investigation: A Guide for First Responders*, July 2001.

[6] Nicole Beebe and Jan Guynes Clark. A hierarchical, objectives-based framework for the digital investigations process. *Digital Investigation*, 2(2):147–167, 2005.

[7] Venansius Baryamureeba and Florence Tushabe. The enhanced digital investigation process model. In *Digital Forensic Research Workshop*, Utica, New York, 2004.

[8] Brian Carrier and Eugene H. Spafford. Getting physical with the digital investigation process. *International Journal of Digital Evidence*, 2(2), 2003.

[9] Carole Fennelly. Analysis: The forensics of internet security. *SunWorld*, July 26, 2000.

[10] Van Jacobson, Craig Leres, and Steven McCanne. Packet capture library. `http://www.tcpdump.org/`. (Last visited: May 26, 2007).

[11] Nate King and Errol Weiss. Analyze this! *Information Security Magazine*, February 2002.

[12] NIKSUN Enterprise. Netdetector: Proactive security surveillance solution. `http://www.niksun.com/`. (Last visited: May 26, 2007).

[13] Sandstorm Enterprises. Netintercept: A network analysis and visibility tool. `http://www.sandstorm.com/`. (Last visited: May 26, 2007).

[14] Teresa Lunt. Detecting intruders in computer systems. In *1993 Conference on Audit and Computer Technology*, 1993.

[15] Kymie Tan, David Thompson, and A. Ruighaver. Intrusion detection systems and a view to its forensic applications. Technical report, Department of Computer Science, University of Melbourne, 1999.

[16] Jim Yuill, S. Wu, Fengmin Gong, and Ming-Yuh Huang. Intrusion detection for on-going attack. *RAID*, 1999.

[17] Srinivas Mukkamala and Andrew Sung. Identifying significant features for network forensic analysis using artificial intelligent techniques. *International Journal of Digital Evidence*, 1(4), winter 2003.

[18] T. Ptacek and T. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. *Secure Networks, Inc.*, January 1998.

[19] Brian Scottberg, William Yurcik, and David Doss. Internet honeypots: Protection or entrapment? In *Proceedings of the IEEE International Symposium on Technology and Society (ISTAS)*, 2002.

[20] Lance Spitzner. The honeynet project. `http://www.honeynet.org`. (Last visited: May 26, 2007).

[21] Keisuke Takemori, Kenji Rikitake, Yutaka Miyake, and Koji Nakao. Intrusion trap system: an efficient platform for gathering intrusion-related information. In *10th International Conference on Telecommunications*, volume 1, pages 614–619, 2003.

[22] Alec Yasinsac and Yanet Manzano. Honeytraps, a network forensic tool. In *Sixth Multi-Conference on Systemics, Cybernetics and Informatics*, 2002.

[23] Barbara Redmon. Maintaining forensic evidence for law enforcement agencies from a federation of decoy networks: An extended abstract. *Mitretek Systems*, Fall 2002.

[24] Stuart Staniford-Chen and L. Todd Heberlein. Holding intruders accountable on the internet. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 39–49, May 1995.

[25] Peter Sommer. Intrusion detection systems as evidence. *Computer Networks*, 31, 1999.

[26] D. Brezinski and T. Killalea. Guidelines for evidence collection and archiving. BCP 55, RFC 3227, February 2002.

[27] A.A.E. Ahmed and Issa Traore. Detecting computer intrusions using behavioral biometrics. In *Third Annual Conference on Privacy, Security and Trust*, St. Andrews, New Brunswick, Canada, October 2005.

[28] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison-Wesley, 2005.

[29] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[30] Honeynet Project. *Know Your Enemy: Sebek*, November 2003.

[31] Martin Roesch and Chris Green. *Snort Users Manual*, April 2003.

[32] Michael Widenius. Mysql database server. `http://www.mysql.com/`. (Last visited: May 26, 2007).

[33] CERT Coordination Center. Acid: Analysis console for intrusion databases. `http://acidlab.sourceforge.net/`. (Last visited: May 26, 2007).

[34] The Chkrootkit Project. Chkrootkit: Locally checks for signs of a rootkit. `http://www.chkrootkit.org/`. (Last visited: May 26, 2007).

[35] Michal Zalewski. P0f v2: A versatile passive os fingerprinting tool. `http://lcamtuf.coredump.cx/p0f.shtml`. (Last visited: May 26, 2007).

[36] Pierre Beyssac. Bing: A point-to-point bandwidth measurement tool. `http://fgouget.free.fr/bing/index-en.shtml`. (Last visited: May 26, 2007).

[37] Clifford Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday Publishing, 1989.

[38] Yin Zhang and Vern Paxson. Detecting stepping stones. In *9th USENIX Security Symposium*, pages 171–184, Aug 2000.

[39] S. C. Lee and C. Shields. Tracing the source of network attack: A technical, legal and societal problem. In *Proceedings of the 2001 IEEE*

Workshop on Information Assurance and Security, pages 239–246, june 2001.

[40] Andrey Belenky and Nirwan Ansari. On ip traceback. *IEEE Communications Magazine*, 41(7):142–153, 2003.

[41] Thomas E. Daniels and Eugene H. Spafford. Network traffic tracking systems: folly in the large? In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 119–124, New York, NY, USA, 2000. ACM Press.

[42] Jon Postel and J. Reynolds. *Telnet Protocol Specification.* RFC 854, May 1983.

[43] B. Kantor. *BSD Rlogin.* RFC 1282, Dec 1991.

[44] C. Lonvick. *SSH Protocol Architecture.* Cisco Systems, Inc., December 2004.

[45] Jon Postel. *Transmission Control Protocol.* RFC 793, sep 1981.

[46] David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson, Jason Coit, and Stuart Staniford. Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In *RAID 2002: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection,*, pages 17–35, october 2002.

[47] Richard W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1994.

[48] Jon Postel and J. Reynolds. *File Transfer Protocol.* RFC 959, Oct 1985.

[49] P.B. Danzig, S. Jamin, R. Caceres, D.J. Mitzel, and D. Estrin. An empirical workload model for driving wide-area tcp/ip network simulations. *Internetworking: Research and Experience*, 3(1):1 – 26, 1992.

[50] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Trans. Netw.*, 3(3):226–244, 1995.

[51] Peter Danzig and Sugih Jamin. tcplib: A library of tcp internetwork traffic characteristic. Technical Report CS-SYS-91-01, Computer Science Department, University of Southern California, 1991.

[52] Kunikazu Yoda and Hiroaki Etoh. Finding a connection chain for tracing intruders. In *ESORICS '00: Proceedings of the 6th European Symposium on Research in Computer Security*, pages 191–205, London, UK, 2000. Springer-Verlag.

[53] Xinyuan Wang, Douglas S. Reeves, and Shyhtsun Felix Wu. Inter-packet delay based correlation for tracing encrypted connections through stepping stones. In *ESORICS '02: Proceedings of the 7th European Symposium on Research in Computer Security*, pages 244–263, London, UK, 2002. Springer-Verlag.

[54] Avrim Blum, Dawn Song, and Shobha Venkataraman. Detection of interactive stepping stones: Algorithms and confidence bounds. In *Lecture Notes in Computer Science*, volume 3224, pages 258–277, Jan 2004.

[55] T. He and L. Tong. A signal processing prospective to stepping-stone detection. In *Conference on Information Sciences and Systems 2006 (CISS'06)*, Princeton, NJ, 2006.

[56] Xinyuan Wang and Douglas S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of interpacket delays. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 20–29, New York, NY, USA, 2003. ACM Press.

[57] Pai Peng, Peng Ning, and Douglas S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *IEEE Symposium on Security and Privacy*, pages 334–349, 2006.

[58] Kwong H. Yung. Detecting long connection chains of interactive terminal sessions. In *RAID 2002, Lecture Notes in Computer Science*, volume 2516, pages 1–16, Jan 2002.

[59] Jianhua Yang and Shou-Hsuan Stephen Huang. A real-time algorithm to detect long connection chains of interactive terminal sessions. In *InfoSecu '04: Proceedings of the 3rd international conference on Information security*, pages 198–203, New York, NY, USA, 2004. ACM Press.

[60] Jianhua Yang and Shou-Hsuan Huang. Matching tcp packets and its application to the detection of long connection chains on the internet. In *AINA 2005: 19th International Conference on Advanced Information Networking and Applications*, pages 1005–1010, March 2005.

[61] Hyung Woo Kang, Soon Jwa Hong, and Dong Hoon Lee. Matching connection pairs. In *Lecture Notes in Computer Science*, volume 3320, pages 642–649, Jan 2004.

[62] Brian Carrier and Clay Shields. The session token protocol for forensics and traceback. *ACM Trans. Inf. Syst. Secur.*, 7(3):333–362, 2004.

[63] Florian Buchholz and Clay Shields. Providing process origin information to aid in network traceback. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[64] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. DIDS (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *Proceedings of the 14th National Computer Security Conference*, pages 167–176, Washington, DC, 1991.

[65] Hyun Tae Jung, Hae Lyong Kim, Yang Min Seo, Ghun Choe, Sang Min, and Chong Sang Kim. Caller identification system in the internet

environment. In *Proceedings of UNIX Security Symposium IV*, pages 69–78, Santa Clara, California, oct 1993.

[66] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the 3rd USENIX UNIX Security Symposium*, pages 85–92, Baltimore, Maryland, 14–16 September 1992. USENIX Association.

[67] M. St. Johns. *Identification Protocol.* RFC 1413, February 1993.

[68] Xinyuan Wang, Douglas S. Reeves, S. Felix Wu, and Jim Yuill. Sleepy watermark tracing: an active network-based intrusion response framework. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 369–384, 2001.

[69] Jianqiang Xin, Linfeng Zhang, Brad Aswegan, John Dickerson, Julie Dickerson, Thomas Daniels, and Yong Guan. A testbed for evaluation and analysis of stepping stone attack attribution techniques. In *Proceedings of TridentCom 2006*, Barcelona, Spain, Mar 2006.

[70] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, 1993.

[71] Sun Microsystems Inc. The j2se development kit. `http://java.sun.com`. (Last visited: May 26, 2007).

[72] Keita Fujii. Jpcap: Java package for packet capture. `http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html`. (Last visited: May 26, 2007).

[73] Ruoming Pang and Vern Paxson. Lbnl-ftp-pkt. `http://www-nrg.ee.lbl.gov/anonymized-traces.html`. (Last visited: May 26, 2007).

[74] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2000.

[75] Steven J. Templeton and Karl Levitt. A requires/provides model for computer attacks. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 31–38. ACM Press, 2000.

[76] Ahmad Almulhem and Issa Traore. Experience with engineering a network forensics system. *Lecture Notes in Computer Science*, 3391:62–71, January 2005.

[77] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002.

[78] William B. Cavnar and John M. Trenkle. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, US, 1994.

[79] Frank Scheelen. Libtextcat: Language guessing library. `http://software.wise-guys.nl/libtextcat/`. (Last visited: May 26, 2007).

[80] A. D. Gordon. *Classification.* Chapman & Hall/CRC, 2nd edition, 1999.